Arquitecturas Basadas en Servicios (Microservicios)

“El diseño arquitectónico de software es la clave para construir sistemas sostenibles.”

— Grady Booch

¿Por qué se necesita el patrón Saga?

- Cada microservicio tiene su propia base de datos.
- La transacción distribuida (ej. actualizar varias bases de datos simultáneamente) no puede utilizar mecanismos tradicionales como **2PC (Two-Phase Commit)** porque estos pueden introducir bloqueos prolongados, reducir escalabilidad y generar fallas en la disponibilidad.
- Por lo tanto, es necesario un mecanismo que garantice **consistencia eventual** sin bloquear los servicios.

¿Cómo funciona el patrón Saga?

Una Saga es una **secuencia de transacciones locales** ejecutadas en diferentes servicios, donde:

1. Cada paso de la Saga realiza una **acción local** (transacción).

2. Si un paso falla, se ejecuta una **compensación** (rollback local).

El patrón Saga utiliza dos enfoques principales para la coordinación:

1. Coreografía (Choreography)

- No hay un coordinador central. Los servicios se comunican de manera **indirecta** mediante **eventos**.
- Cada servicio publica un evento cuando completa su transacción local, y otros servicios suscritos a este evento realizan su acción correspondiente.
- Si ocurre un fallo, los servicios ejecutan **acciones compensatorias** publicando eventos de rollback.

Ejemplo de flujo con coreografía:

Supongamos una **reserva de viaje** que incluye:

1. Reservar vuelo
2. Reservar hotel
3. Realizar pago

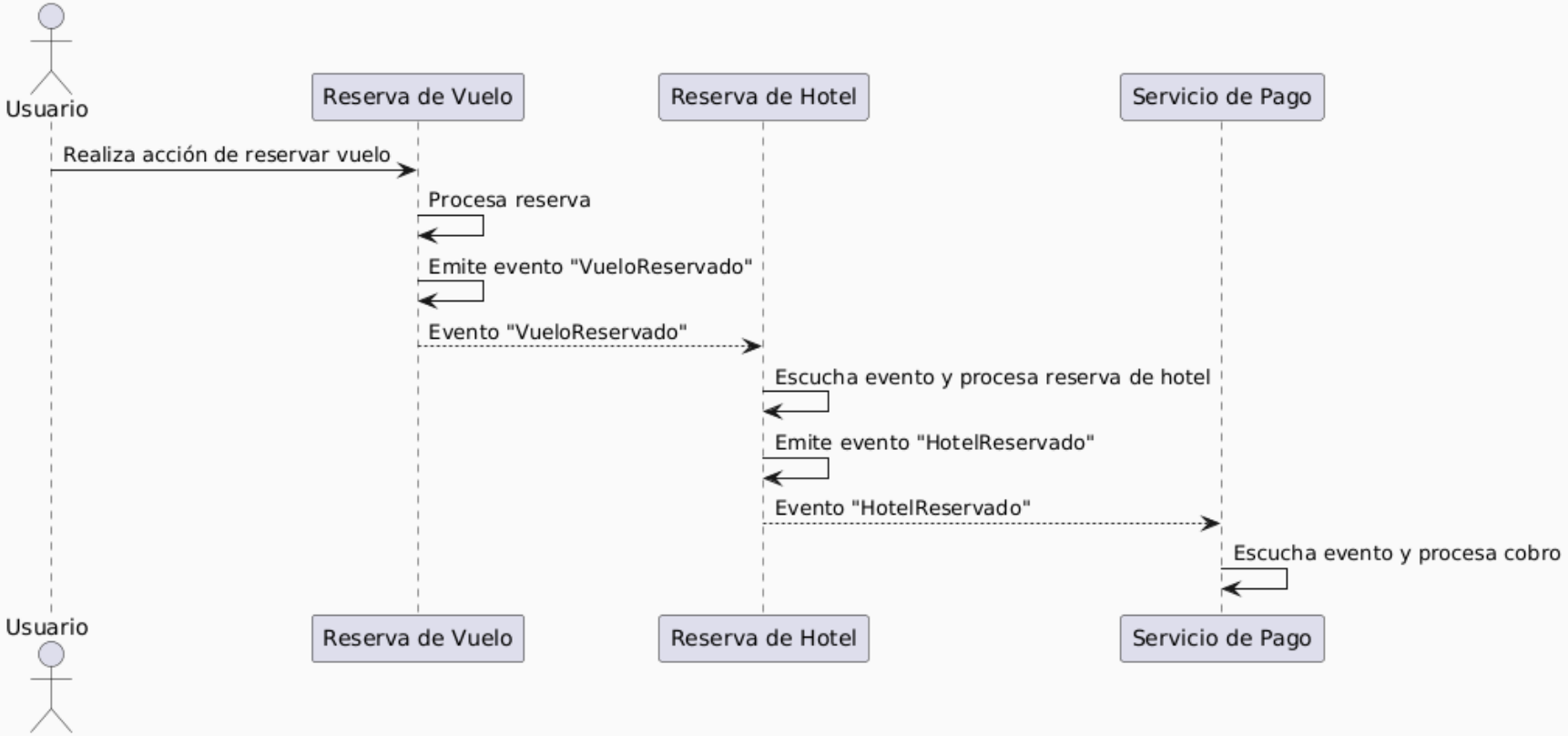
Ejemplo de flujo con coreografía:

Flujo:

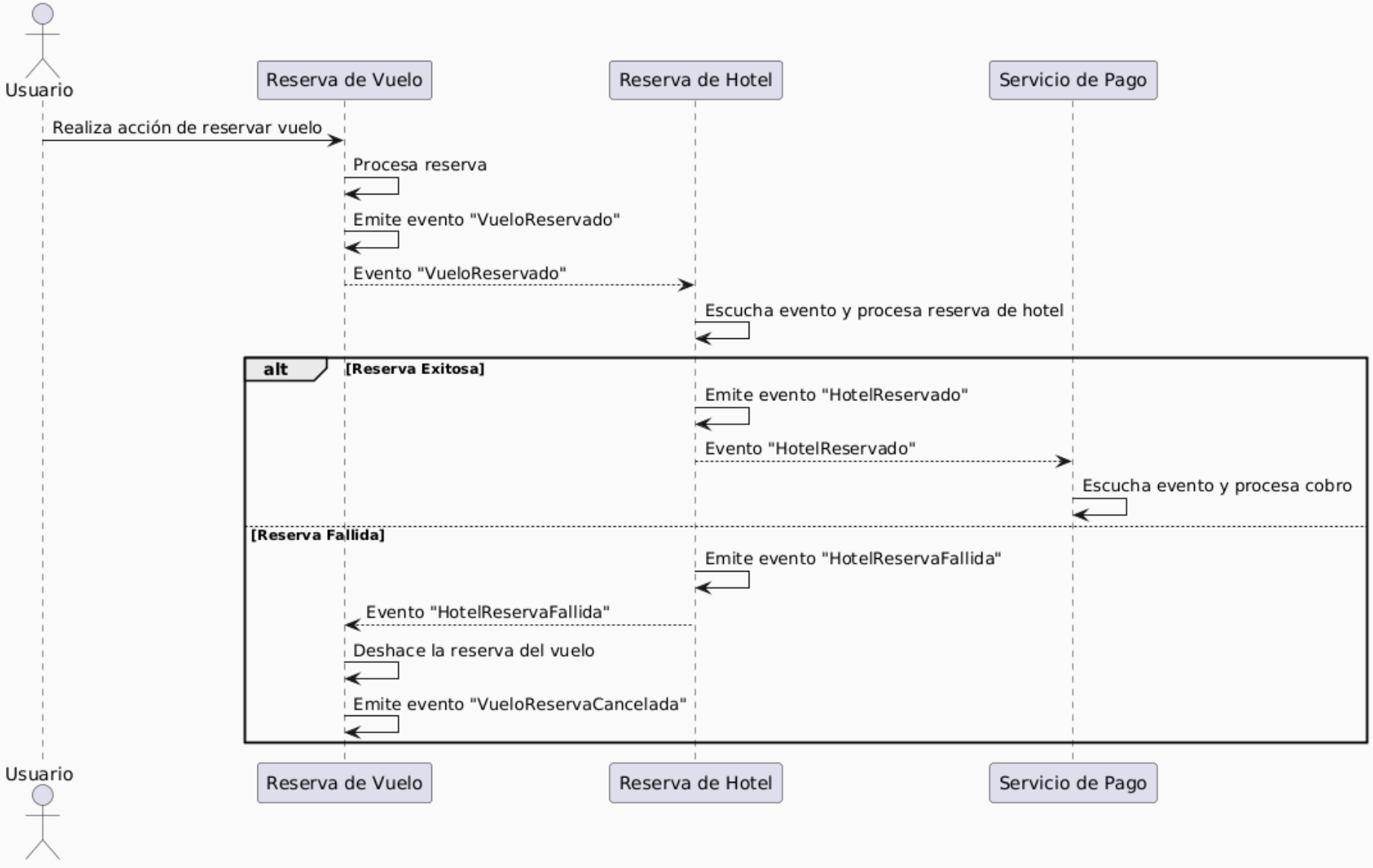
1. El servicio **“reserva de vuelo”** realiza su acción y emite un evento **“VueloReservado”**.

2. El servicio **“reserva de hotel”** escucha el evento y realiza su acción, emitiendo un evento **“HotelReservado”**.

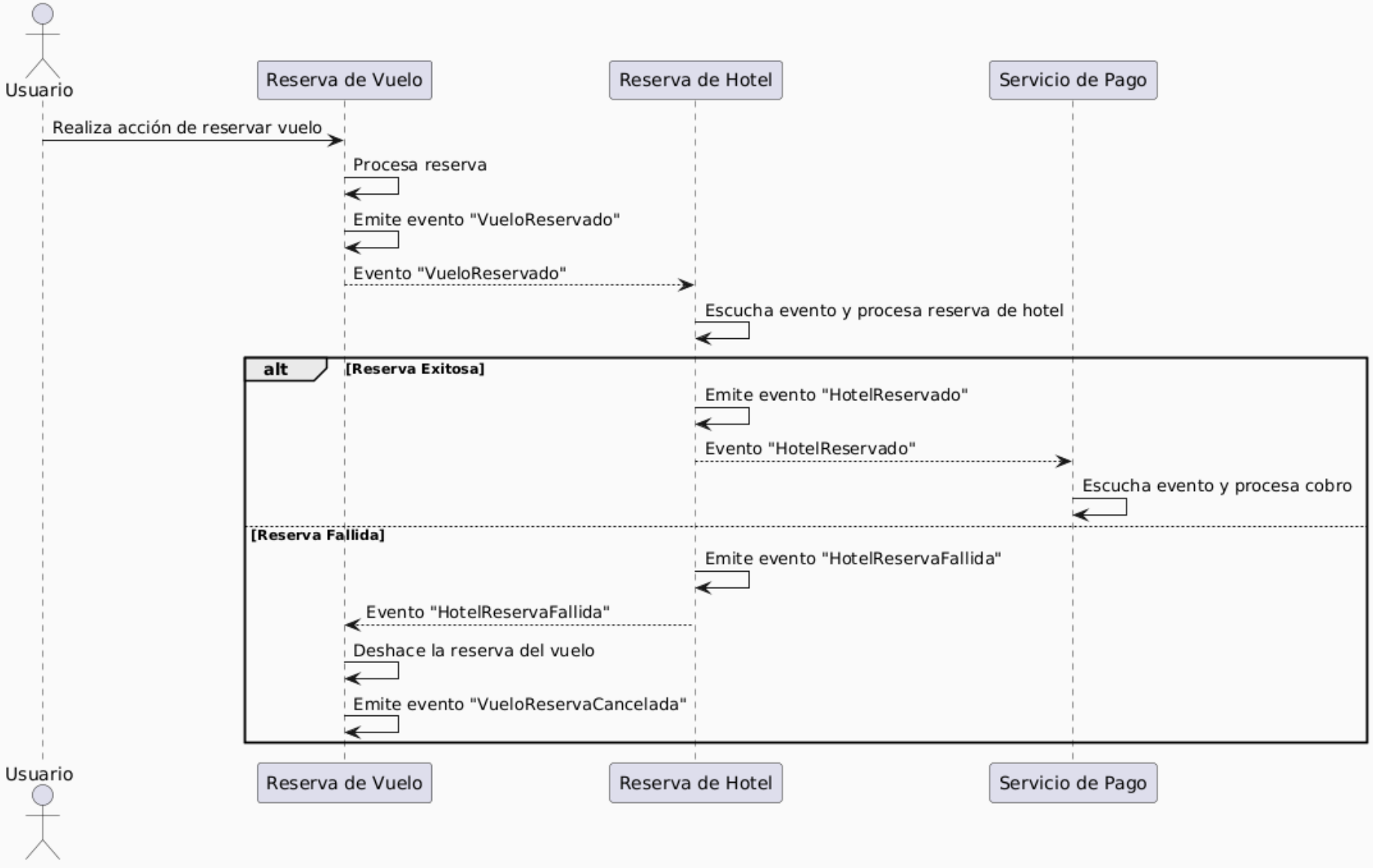
3. El servicio **“pago”** escucha el evento **“HotelReservado”** y realiza la acción de cobro.



Ejemplo de flujo con coreografía:



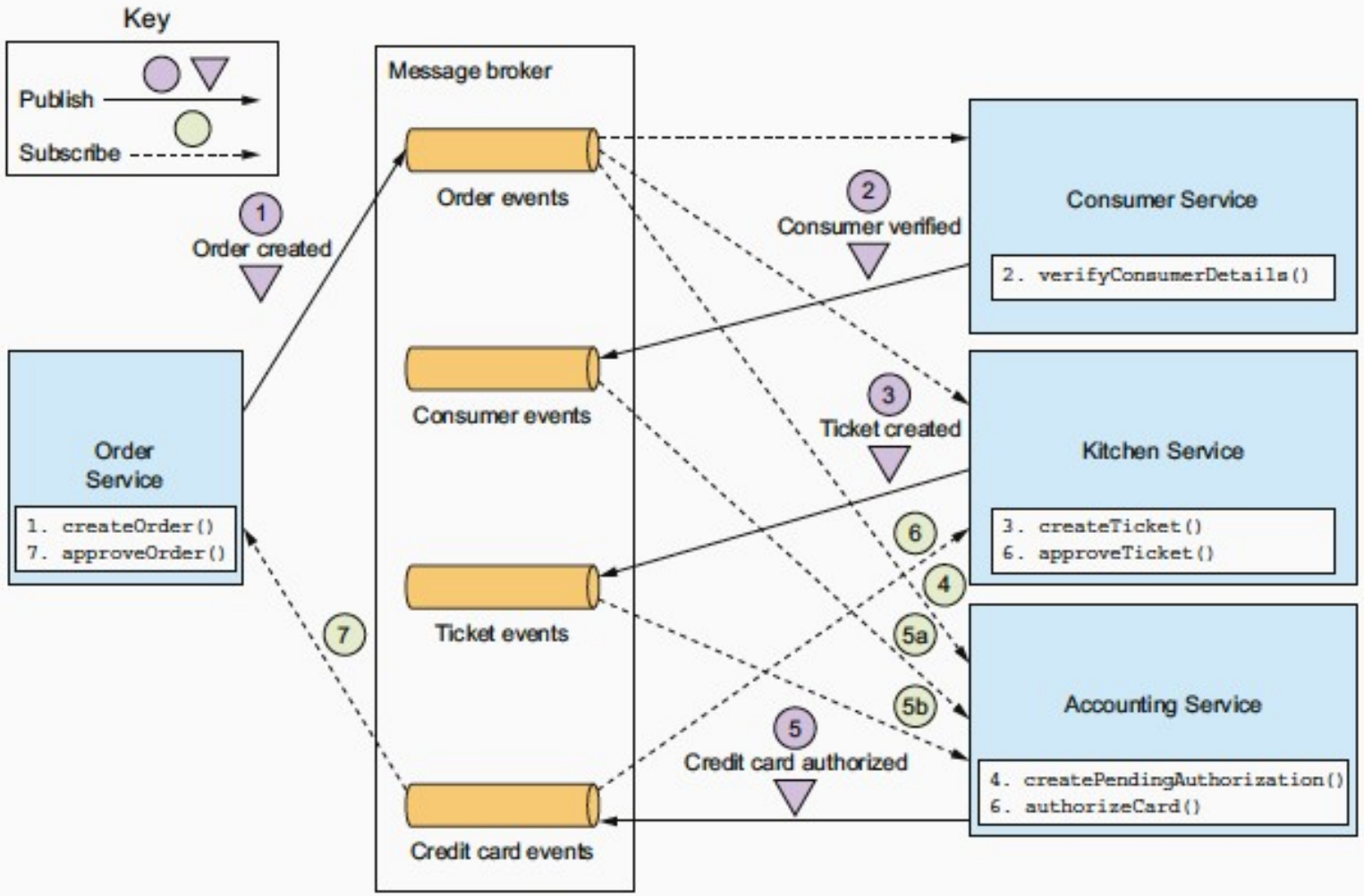
Ejemplo de flujo con coreografía:

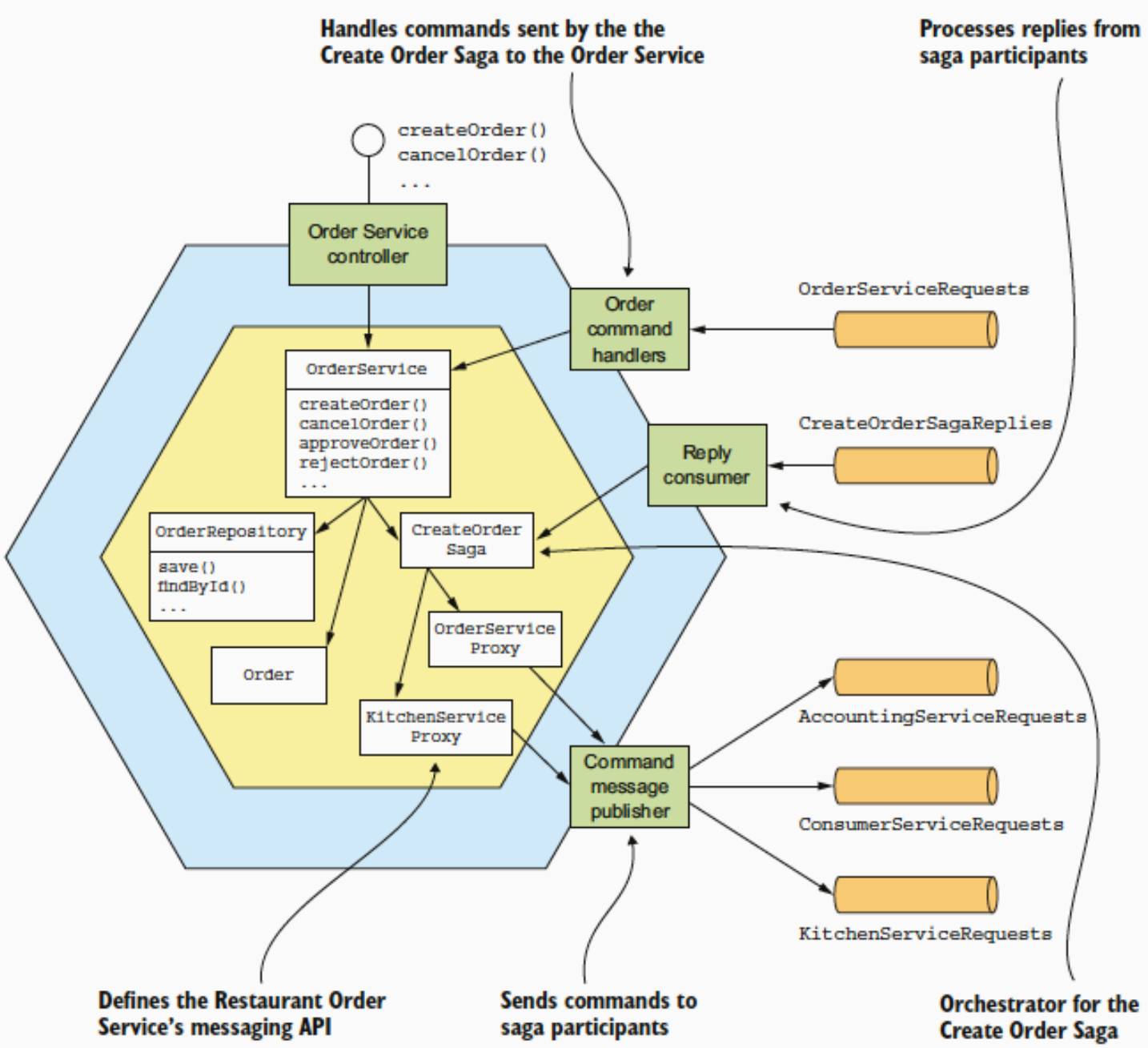


Ejemplo de flujo con coreografía:

```python
producer = Producer(producer_conf)
consumer = Consumer(consumer_conf)
consumer.subscribe(['hotel_reserva_fallida'])

def reservar_vuelo():
    print("🚀 Reservando vuelo...")
    time.sleep(2)  # Simula tiempo de procesamiento
    print("✅ Vuelo reservado.")

    # Publica el evento de éxito
    producer.produce('vuelo_reservado', key="vuelo", value="VueloReservado")
    producer.flush()
    print("📢 Evento 'VueloReservado' publicado.")

def cancelar_vuelo():
    print("⚠️ Recibido evento 'HotelReservaFallida'")
    print("❌ Cancelando reserva de vuelo...")
    time.sleep(2)
    print("✅ Vuelo cancelado.")

    # Publica el evento de cancelación
    producer.produce('vuelo_reserva_cancelada', key="vuelo", value="VueloReservaCancelada")
    producer.flush()
    print("📢 Evento 'VueloReservaCancelada' publicado.")
```

Coregrafi


Key
Publish
Subscribe
Message broker
Order events
Consumer events
Ticket events
Credit card events
Order Service
1. createOrder()
7. approveOrder()
1
Order created
2
Consumer verified
Consumer Service
2. verifyConsumerDetails()
3
Ticket created
Kitchen Service
3. createTicket()
6. approveTicket()
6
4
5a
5b
7
5
Credit card authorized
Accounting Service
4. createPendingAuthorization()
6. authorizeCard()



Preparando Microservicios

Orquestación (Orchestration)

- Existe un **coordinador central** llamado **Orquestador de Saga**, que administra la ejecución de los pasos.
- El orquestador envía comandos a los servicios para ejecutar transacciones locales y decide qué hacer si ocurre un error.

Orquestación (Orchestration)

Ejemplo de flujo con orquestación:

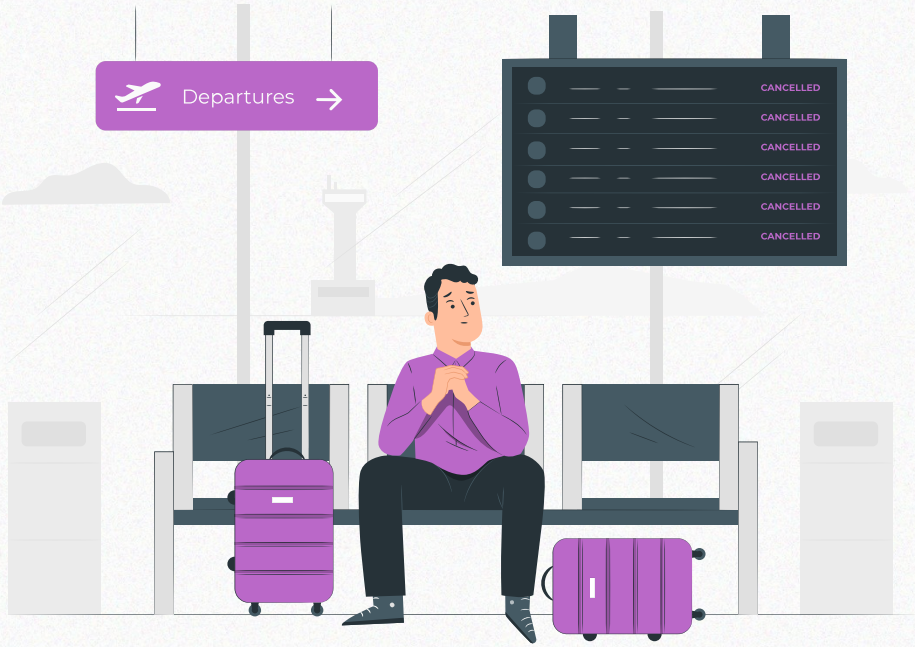
1. El orquestador inicia la Saga enviando un comando **“ReservarVuelo”** al servicio de vuelos.

2. Al completarse exitosamente, el orquestador envía un comando **“ReservarHotel”** al servicio de hoteles.

3. Si todo funciona, se envía un comando **“RealizarPago”** al servicio de pagos.

4. Si falla algún paso, el orquestador envía comandos de **compensación**:

• Si la reserva del hotel falla, el orquestador envía **“CancelarVuelo”** al servicio de vuelos.



CODIGO
powered by Tecsup/>

Conductor

Apache
Airflow

zeebe
by Camunda

Netflix Conductor

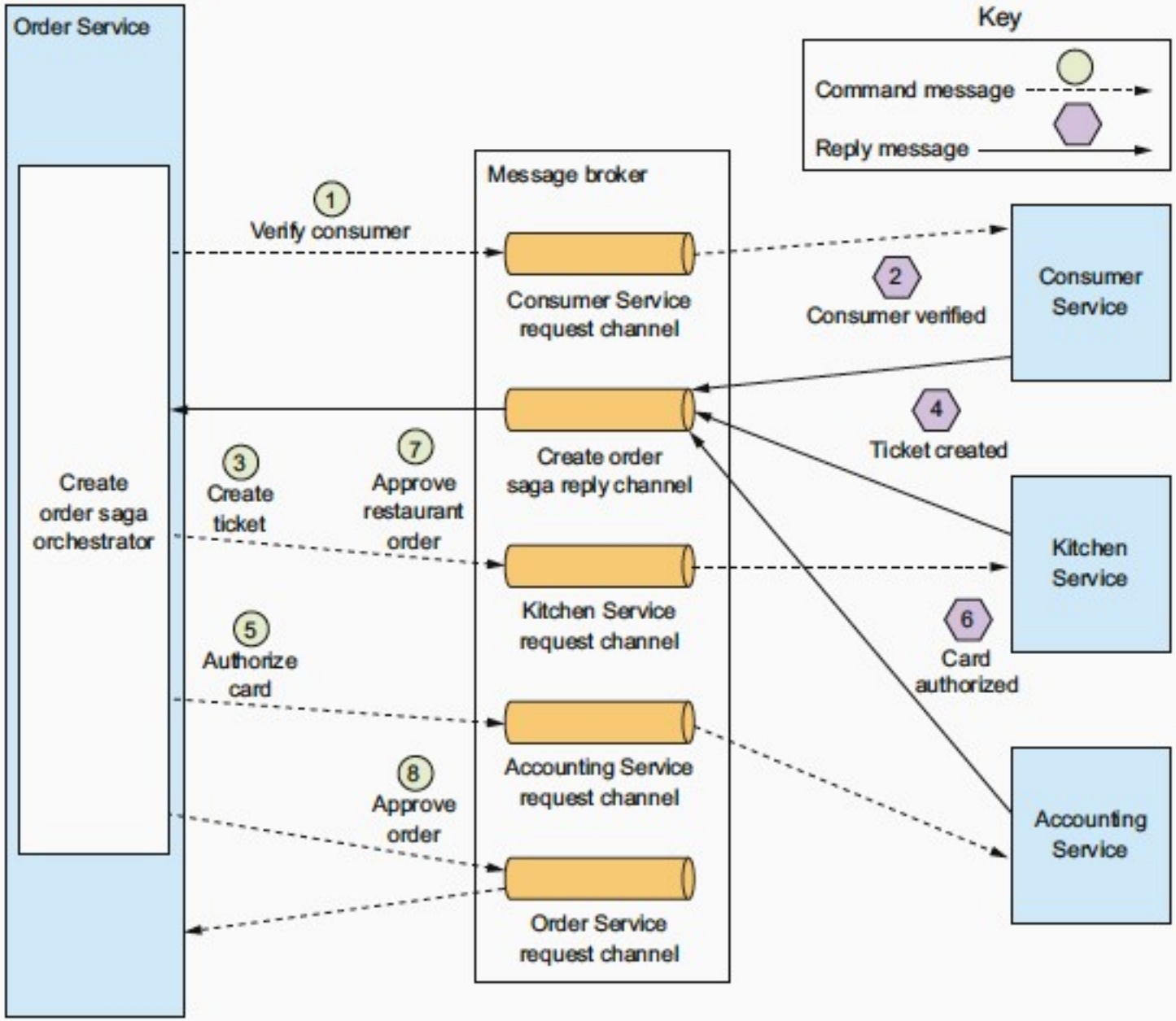
1. **Workflow**:
   - Es el flujo general de tareas que debes ejecutar.
   - Se define en formato **JSON** o **YAML**.
2. **Task**:
   - Cada paso dentro del flujo de trabajo.
   - Puede ser de varios tipos:
     - **HTTP_TASK**: Llama a un endpoint HTTP.
     - **SIMPLE_TASK**: Tarea básica manejada por el worker.
     - **EVENT_TASK**: Publica o escucha eventos.
     - **SUB_WORKFLOW**: Permite anidar otros flujos de trabajo.
3. **Worker**:
   - Es el componente que ejecuta las tareas definidas en el workflow.
   - Los workers interactúan con Netflix Conductor a través de su **API REST**.
4. **Orquestador**:
   - Netflix Conductor actúa como **orquestador central**, coordinando las tareas del workflow.

```
{
  "name": "travel_booking_workflow",
  "description": "Workflow to book a flight, hotel, and process payment with error handling",
  "version": 1,
  "tasks": [
    {
      "name": "reserve_flight_task",
      "taskReferenceName": "reserve_flight",
      "type": "HTTP",
      "httpRequest": {
        "method": "POST",
        "uri": "http://localhost:5001/reserve_flight"
      },
      "retryCount": 2,
      "retryLogic": "FIXED",
      "retryDelaySeconds": 5
    },
    {
      "name": "reserve_hotel_task",
      "taskReferenceName": "reserve_hotel",
      "type": "HTTP",
      "httpRequest": {
        "method": "POST",
        "uri": "http://localhost:5002/reserve_hotel"
      },
      "retryCount": 2,
      "retryLogic": "FIXED",
      "retryDelaySeconds": 5
    }
```

```
    {
      "name": "reserve_hotel_with_fallback",
      "taskReferenceName": "reserve_hotel_with_fallback",
      "type": "DECISION",
      "decisionCases": {
        "FAILED": [
          {
            "name": "cancel_flight_task",
            "taskReferenceName": "cancel_flight",
            "type": "HTTP",
            "httpRequest": {
              "method": "POST",
              "uri": "http://localhost:5001/cancel_flight"
            }
          }
        ]
      },
      "defaultCase": [
        {
          "name": "process_payment_task",
          "taskReferenceName": "process_payment",
          "type": "HTTP",
          "httpRequest": {
            "method": "POST",
            "uri": "http://localhost:5003/process_payment"
          }
        }
      ]
    }
  ],
  "schemaVersion": 2
}
```

Puedes construir tu propio orquestador

```python
class Orquestador:
    def __init__(self):
        self.vuelo_url = "http://localhost:5001/reservar_vuelo"
        self.hotel_url = "http://localhost:5002/reservar_hotel"
        self.cancelar_vuelo_url = "http://localhost:5001/cancelar_vuelo"

    def ejecutar(self):
        try:
            # Paso 1: Reservar vuelo
            print("🚀 Iniciando reserva de vuelo...")
            response = requests.post(self.vuelo_url)
            if response.status_code != 200:
                raise Exception("Error al reservar vuelo")
            print("✅ Vuelo reservado.")


            # Paso 2: Reservar hotel
            print("🏨 Iniciando reserva de hotel...")
            response = requests.post(self.hotel_url)
            if response.status_code != 200:
                raise Exception("Error al reservar hotel")
            print("✅ Hotel reservado.")


            print("💰 Flujo completado exitosamente.")
        except Exception as e:
            print(f"❌ Error: {str(e)}")
            print("🔄 Ejecutando acciones compensatorias...")
            self.compensar_vuelo()
```


Order Service
Create order saga orchestrator
Key
Command message
Reply message
Message broker
1
Verify consumer
Consumer Service request channel
2
Consumer verified
Consumer Service
Create order saga reply channel
4
Ticket created
3
Create ticket
7
Approve restaurant order
Kitchen Service request channel
Kitchen Service
5
Authorize card
6
Card authorized
Accounting Service request channel
8
Approve order
Accounting Service
Order Service request channel

Implementando la Logica del Negocio con Microservicios

¿Qué es la lógica de negocio?

La lógica de negocio es el conjunto de reglas, procesos y flujos que determinan cómo un sistema debe comportarse para cumplir los objetivos y necesidades de la organización.

1. **Lógica local** (interna al microservicio):
    - Es lógica específica de un microservicio.
2. **Lógica global** (coordinada entre varios servicios):
    - Son flujos de negocio que requieren interacciones entre múltiples servicios. Ejemplo: Procesar una orden en un e-commerce implica inventario, pagos, y envíos.

Diseño de la Lógica Global

Escenario:

Un cliente realiza un pedido que requiere:

1. Validar la orden (servicio de órdenes).
2. Reservar el stock (servicio de inventario).
3. Procesar el pago (servicio de pagos).
4. Generar el envío (servicio de envíos).
5. Si falla un paso, revertir los pasos anteriores.

Diseño con Coreografia

Flujo Basado en Eventos

- **Evento** `OrderCreated`:
  - El servicio de órdenes publica este evento.
  - El servicio de inventario lo escucha y reserva el stock.
- **Evento** `StockReserved`:
  - El servicio de pagos lo escucha y procesa el pago.
- **Evento** `PaymentProcessed`:
  - El servicio de envíos lo escucha y genera el envío.

Diseño con Orquestación

El flujo se modela como pasos secuenciales:

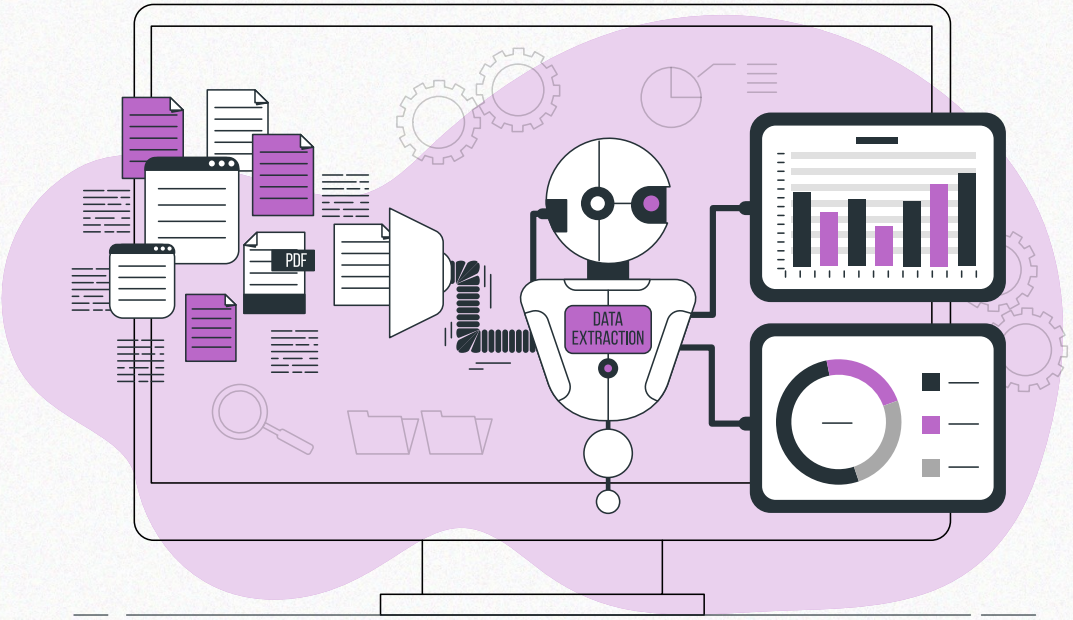
1. Crear la orden (orden-service).
2. Reservar el stock (inventory-service).
3. Procesar el pago (payment-service).
4. Generar el envío (shipping-service).
5. Manejar errores: Si falla el paso 2 o 3, cancelar la orden.

Comparación

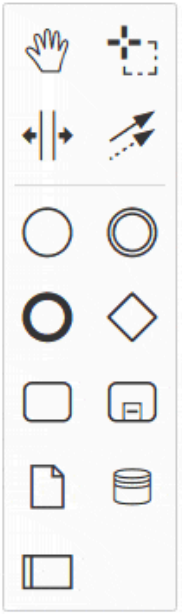
Aspecto	Orquestación	Coreografía
Coordinación	Centralizada (Orquestador).	Descentralizada (Eventos).
Escalabilidad	Puede ser un cuello de botella.	Altamente escalable.
Complejidad	Centralizada en el orquestador.	Distribuida entre servicios.
Tolerancia a Fallos	Fácil de manejar con lógica centralizada.	Difícil de rastrear y depurar.

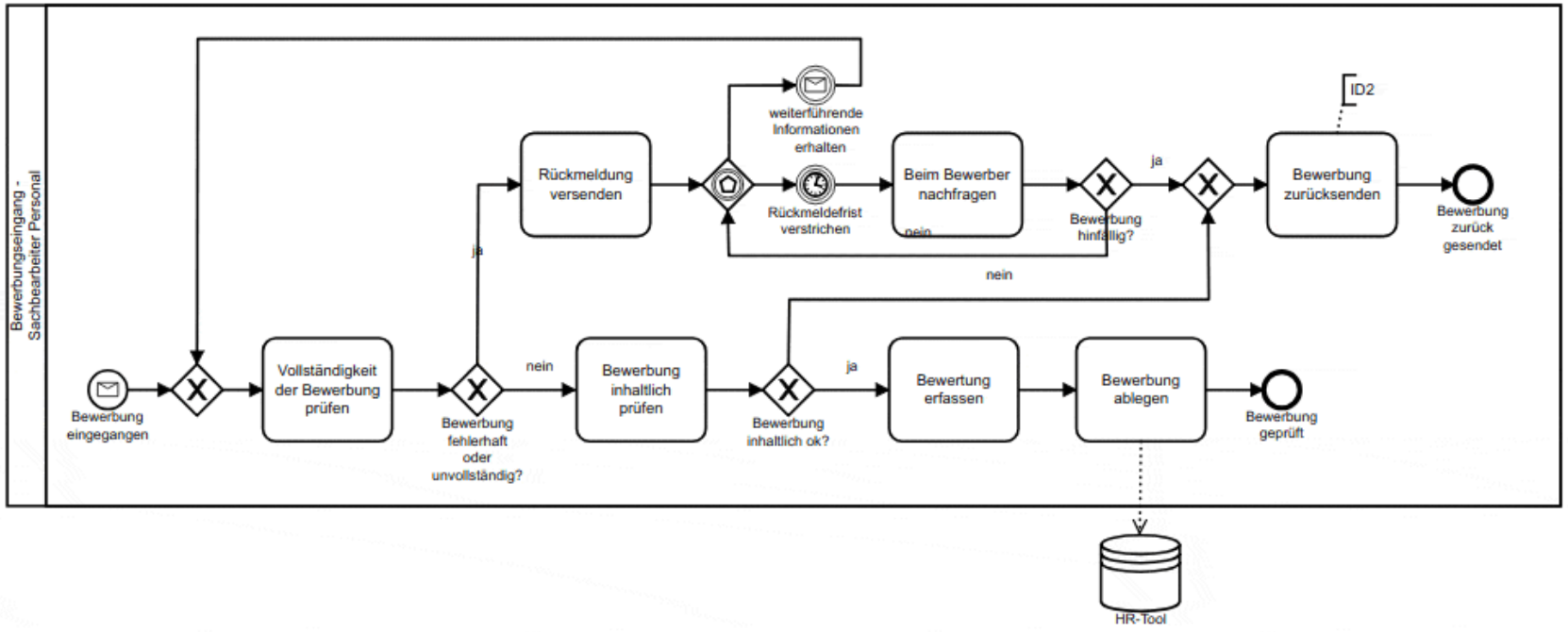
¿Qué es BPMN?

- **BPMN** es un estándar gráfico para modelar procesos de negocio.
- Permite representar procesos mediante **diagramas visuales** que incluyen actividades, decisiones, eventos, y flujos.
- Es **independiente de la implementación**, pero puede ejecutarse usando motores de procesos compatibles, como **Camunda, Zeebe, Activiti**, entre otros.



¿Qué es BPMN?




Bewerbungseingang - Sachbearbeiter Personal
Bewerbung eingegangen
Vollständigkeit der Bewerbung prüfen
Bewerbung fehlerhaft oder unvollständig?
ja
nein
Rückmeldung versenden
weiterführende Informationen erhalten
Rückmeldefrist verstrichen
Beim Bewerber nachfragen
nein
Bewerbung hinfällig?
ja
nein
Bewerbung inhaltlich prüfen
Bewerbung inhaltlich ok?
ja
Bewertung erfassen
Bewerbung ablegen
Bewerbung geprüft
ID2
Bewerbung zurücksenden
Bewerbung zurück gesendet
HR-Tool

Ventajas de Usar BPMN en Orquestadores

1. **Visualización**: Facilita la comprensión de la lógica global mediante diagramas claros.

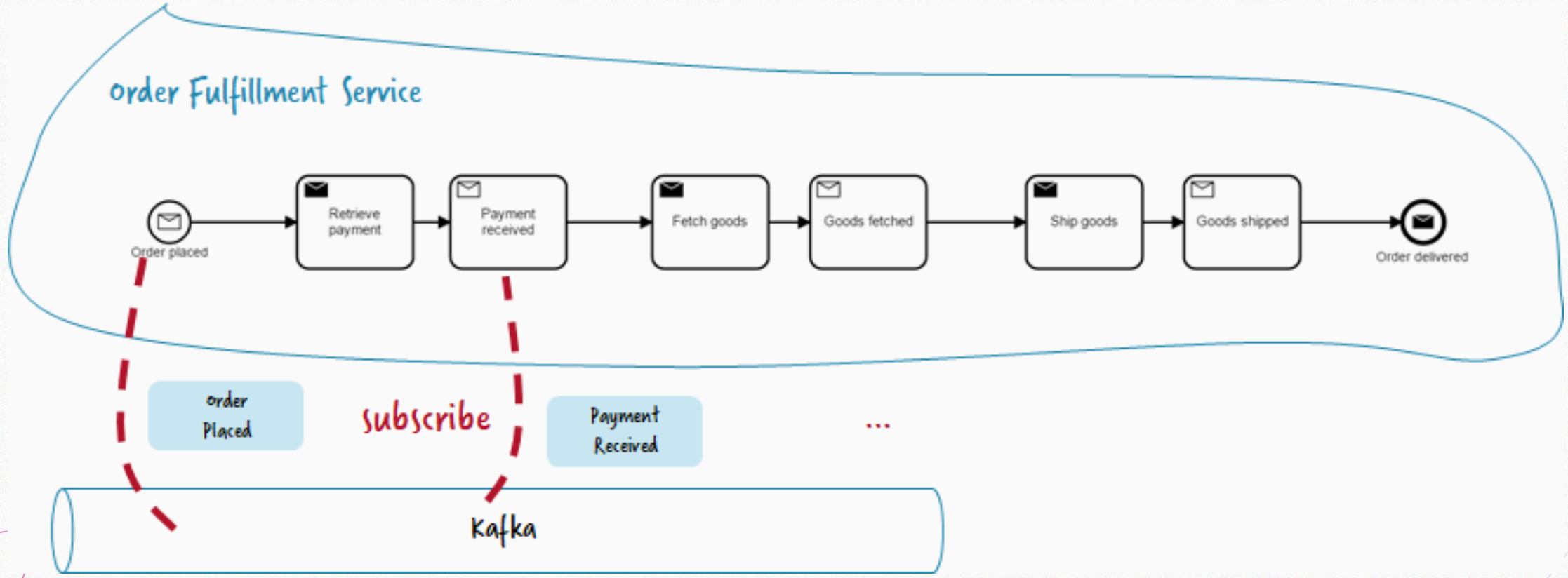
2. **Estandarización**: BPMN es un estándar adoptado internacionalmente.

3. **Ejecutabilidad**: Herramientas como **Camunda** o **Zeebe** pueden **interpretar y ejecutar** directamente los diagramas BPMN.

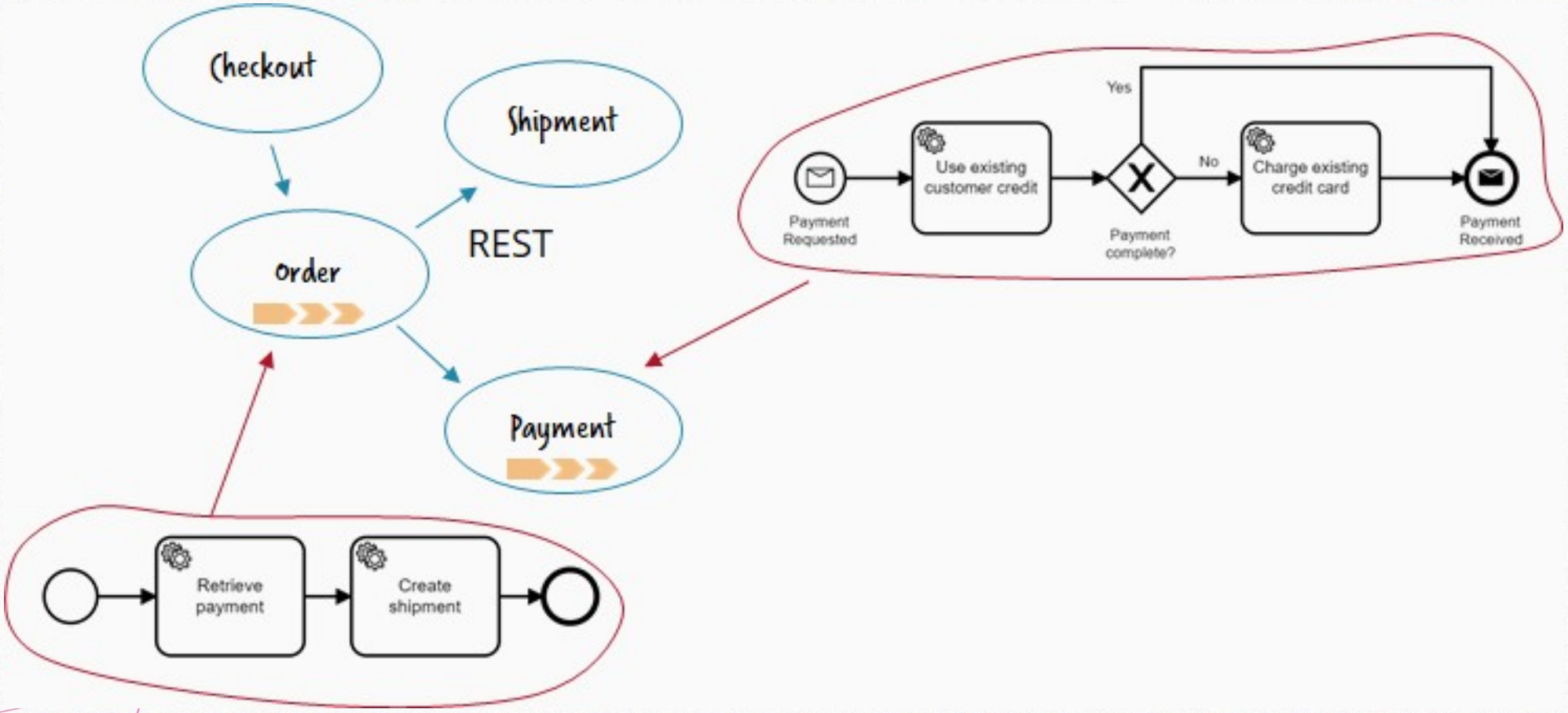
4. **Manejo de errores**: BPMN permite definir flujos de compensación y excepciones (similar al patrón Saga).

5. **Extensibilidad**: Permite integrar microservicios vía **HTTP, eventos** o **colas de mensajes**.

Zeebe


Order Fulfillment Service
Order placed
Retrieve payment
Payment received
Fetch goods
Goods fetched
Ship goods
Goods shipped
Order delivered
Order Placed
subscribe
Payment Received
...
Kafka

Zeebe


Checkout
Shipment
Order
REST
Payment
Retrieve payment
Create shipment
Payment Requested
Use existing customer credit
Yes
No
Payment complete?
Charge existing credit card
Payment Received

```xml
<?xml version="1.0" encoding="UTF-8"?>
<definitions xmlns="http://www.omg.org/spec/BPMN/20100524/MODEL" id="order_process"
             targetNamespace="http://camunda.org/examples">
  <process id="process_order" name="Order Process" isExecutable="true">
    <!-- Inicio -->
    <startEvent id="start_event" name="Inicio del Proceso" />
    <!-- Crear Orden -->
    <serviceTask id="create_order" name="Crear Orden" camunda:delegateExpression="$
{createOrderDelegate}" />
    <!-- Reservar Inventario -->
    <serviceTask id="reserve_inventory" name="Reservar Inventario" camunda:delegateExpression="$
{reserveInventoryDelegate}" />
    <boundaryEvent id="inventory_error" attachedToRef="reserve_inventory">
      <errorEventDefinition errorRef="Error_ReservarInventario" />
      <serviceTask id="cancel_order" name="Cancelar Orden" camunda:delegateExpression="$
{cancelOrderDelegate}" />
    </boundaryEvent>
    <!-- Procesar Pago -->
    <serviceTask id="process_payment" name="Procesar Pago" camunda:delegateExpression="$
{processPaymentDelegate}" />
    <!-- Generar Envío -->
    <serviceTask id="generate_shipping" name="Generar Envío" camunda:delegateExpression="$
{generateShippingDelegate}" />
    <!-- Fin -->
    <endEvent id="end_event" name="Fin del Proceso" />
    <!-- Flujos -->
    <sequenceFlow sourceRef="start_event" targetRef="create_order" />
    <sequenceFlow sourceRef="create_order" targetRef="reserve_inventory" />
    <sequenceFlow sourceRef="reserve_inventory" targetRef="process_payment" />
    <sequenceFlow sourceRef="process_payment" targetRef="generate_shipping" />
    <sequenceFlow sourceRef="generate_shipping" targetRef="end_event" />
    <sequenceFlow sourceRef="inventory_error" targetRef="cancel_order" />
    <sequenceFlow sourceRef="cancel_order" targetRef="end_event" />
  </process>
</definitions>
```

CODIGO
powered by Tecsup/>

Flowable
Intelligent Business Automation

zeebe
by Camunda

Activiti™

Comparación

Herramienta	Soporte BPMN	Escalabilidad	Integración	Uso Ideal
Camunda BPM	Sí	Alta (en producción)	REST, Kafka, RabbitMQ	Procesos empresariales complejos
Zeebe	Sí	Altamente escalable	Kafka, gRPC	Microservicios y eventos
Activiti	Sí	Moderada	Java/Spring Boot	Aplicaciones ligeras Java
Flowable	Sí	Alta	REST, tareas asíncronas	Procesos dinámicos complejos
Airflow	No (DAGs)	Alta	Python, REST, sistemas ETL	Procesos ETL y automatización
Netflix Conductor	No (JSON/YAML)	Alta	HTTP, gRPC, Kafka	Orquestación de microservicios
Temporal.io	No	Altamente escalable	Go/Java APIs	Transacciones distribuidas

¿Qué sucede cuando tienes 5 microservicios con sus propios datos, pero necesitas mostrar todo junto en una única página web o app móvil? ¿Quién debería unificarlos?

“¿El frontend?”
“¿Cada microservicio responde con más info?”
“¿Creamos un servicio adicional?”

API Composition Pattern

¿Cómo unimos las piezas?

En una arquitectura de microservicios, **cada servicio maneja una parte específica de los datos**. Sin embargo, cuando un cliente necesita información **agregada** que involucra múltiples servicios, surgen desafíos:

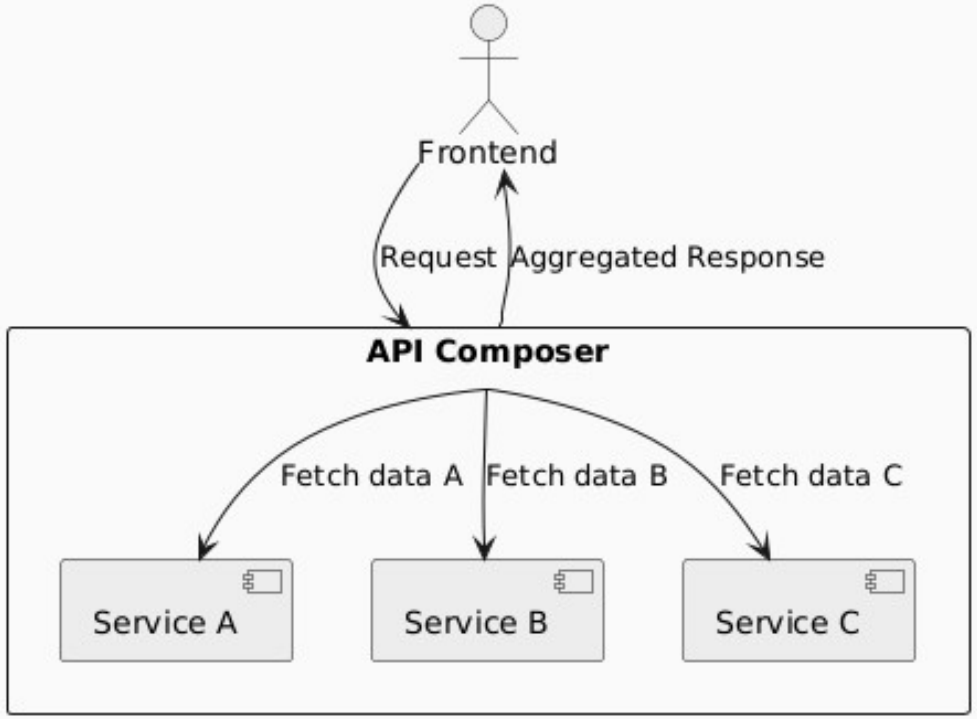
- **Latencia**: El cliente tendría que realizar múltiples llamadas a diferentes servicios.
- **Complejidad**: El cliente tendría que conocer la estructura y detalles de todos los servicios subyacentes.

API Composition Pattern

¿Cómo unimos a muchas APIs?

Un **API Composer** es un servicio que:

- Se encarga de hacer llamadas a varios microservicios.
- **Combina las respuestas** en un único payload.
- Se expone como **una única API** al cliente (frontend, móvil, etc.).



¿Cómo funciona?

1. **API Composition Layer**:

• Actúa como una **capa intermedia** entre el cliente y los microservicios.

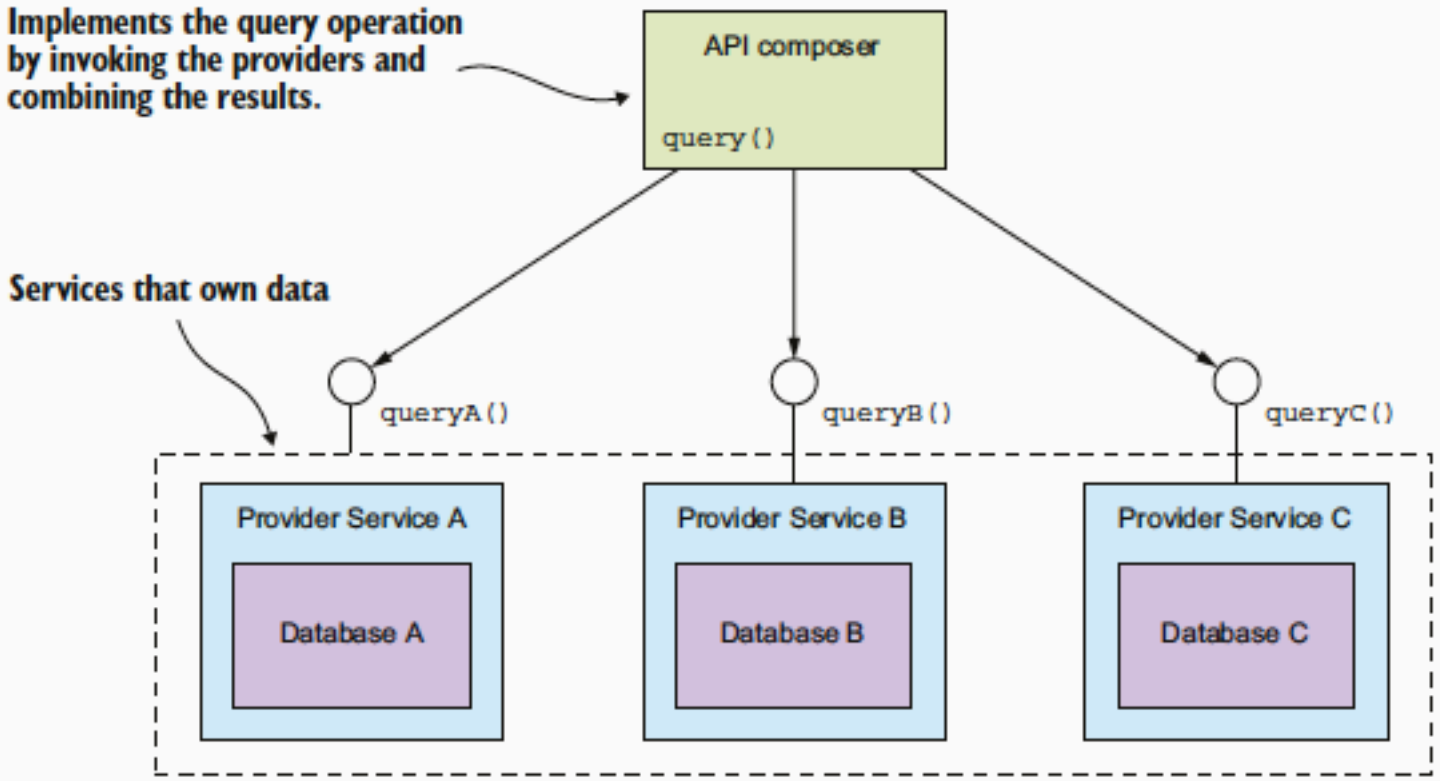
• Realiza llamadas a **múltiples microservicios**, agrega (o compone) los resultados y devuelve una **respuesta consolidada** al cliente.

2. **Agregación de datos**:

• La composición puede hacerse en **tiempo real** o mediante **preagregación** (si los datos no cambian frecuentemente).

3. **Simplificación del cliente**:

• El cliente solo necesita interactuar con una única API, sin preocuparse por la complejidad interna de los microservicios.



¿Cómo funciona?

```
{
  "orderId": "12345",
  "orderDate": "2024-05-01",
  "customer": {
    "name": "Juan Pérez",
    "address": "Av. Siempre Viva 123"
  },
  "products": [
    { "name": "Laptop", "price": 1200 },
    { "name": "Mouse", "price": 20 }
  ],
  "shippingStatus": "En camino"
}
```

Order Service

User Service

Product Service

Shipping Service

¿Cómo funciona?

```
// Endpoint de API Composition
app.get('/orderDetails/:orderId', async (req, res) => {
    const orderId = req.params.orderId;
    try {
        // Llamadas paralelas a los microservicios
        const orderResponse = await axios.get(`${ORDER_SERVICE_URL}/${orderId}`);
        const userResponse = await axios.get(`${USER_SERVICE_URL}/${orderResponse.data.userId}`);

        // Obtener detalles de productos en paralelo
        const productPromises = orderResponse.data.productIds.map(productId =>
            axios.get(`${PRODUCT_SERVICE_URL}/${productId}`)
        );
        const productResponses = await Promise.all(productPromises);

        // Componer la respuesta consolidada
        const composedResponse = {
            orderId: orderResponse.data.id,
            orderDate: orderResponse.data.date,
            total: orderResponse.data.total,
            customer: userResponse.data,
            products: productResponses.map(response => response.data),
        };

        res.json(composedResponse);
    } catch (error) {
        console.error('Error en la composición de la API:', error.message);
        res.status(500).json({ error: 'Error al obtener los detalles de la orden' });
```

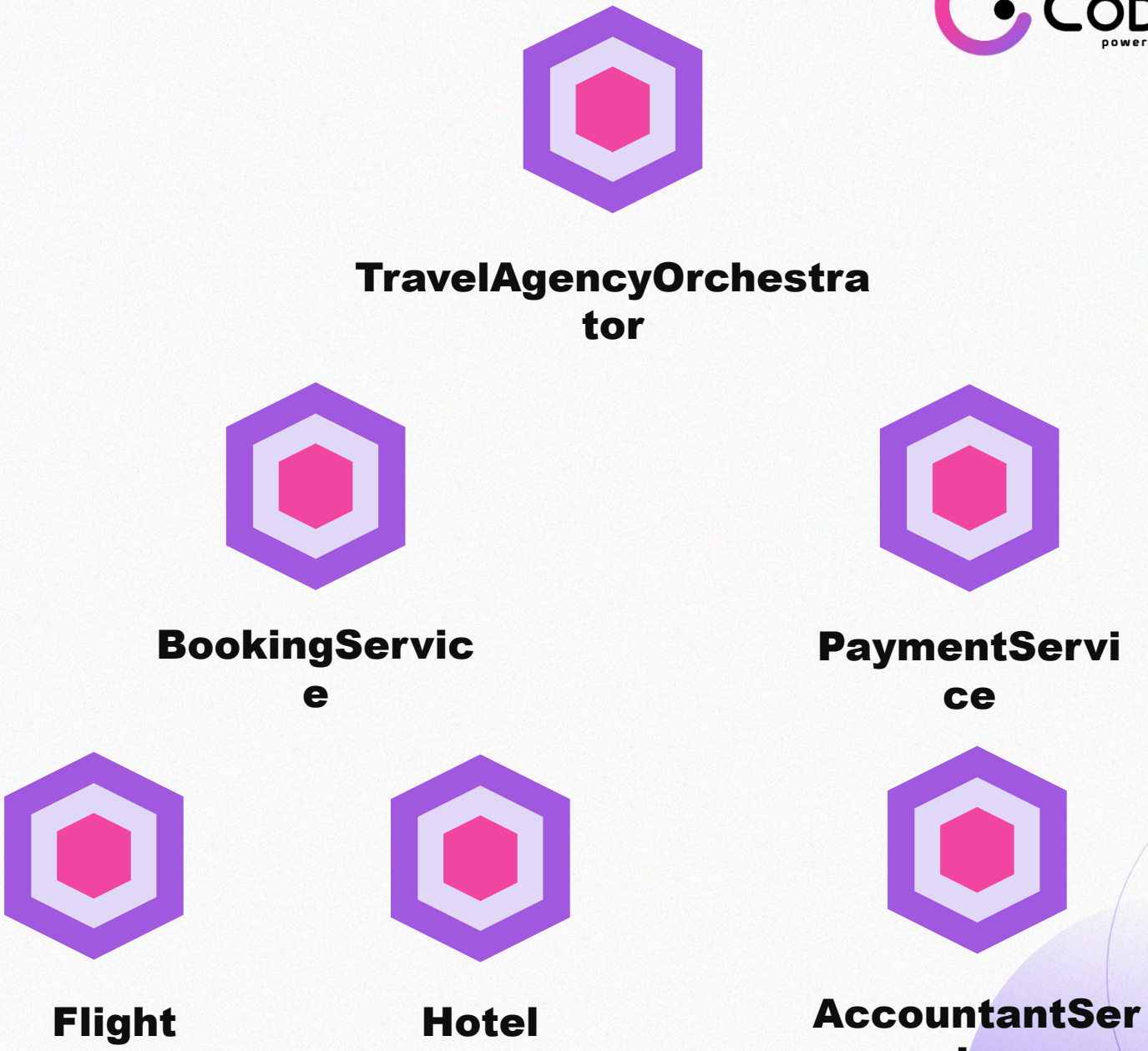
¿Cómo funciona?

```
const express = require('express');
const app = express();
const PORT = 6000;

app.get('/users/:userId', (req, res) => {
    const users = {
        '100': { id: '100', name: 'Juan Pérez', address: 'Av. Siempre Viva 123' },
        '101': { id: '101', name: 'María García', address: 'Calle Falsa 456' },
    };
    res.json(users[req.params.userId]);
});

app.listen(PORT, () => console.log(`User Service escuchando en http://localhost:${PORT}`));
```

¿Cómo funciona?

```
const express = require('express');
const app = express();
const PORT = 6000;

app.get('/users/:userId', (req, res) => {
    const users = {
        '100': { id: '100', name: 'Juan Pérez', address: 'Av. Siempre Viva 123' },
        '101': { id: '101', name: 'María García', address: 'Calle Falsa 456' },
    };
    res.json(users[req.params.userId]);
});

app.listen(PORT, () => console.log(`User Service escuchando en http://localhost:${PORT}`));
```

Arquitectura en capas para Microservicios?



GraphQL

Aunque GraphQL no es estrictamente una herramienta de composición, funciona como un API Gateway que permite **agregar datos** de múltiples microservicios en un único esquema unificado.

Características:

- Define un **esquema único** que abstrae múltiples microservicios.
- Ejecuta **consultas dinámicas** que componen datos en tiempo real.
- **Optimiza las llamadas:** el cliente solicita solo los datos que necesita.

Netflix Zuul

Netflix Zuul es un **API Gateway** que también puede implementar el patrón **API Composition**. Aunque requiere un poco más de configuración en comparación con herramientas como Kong, ofrece flexibilidad y control total, especialmente cuando se integra con aplicaciones basadas en **Spring Boot**.

NETFLIX
ZUUL

Netflix Zuul

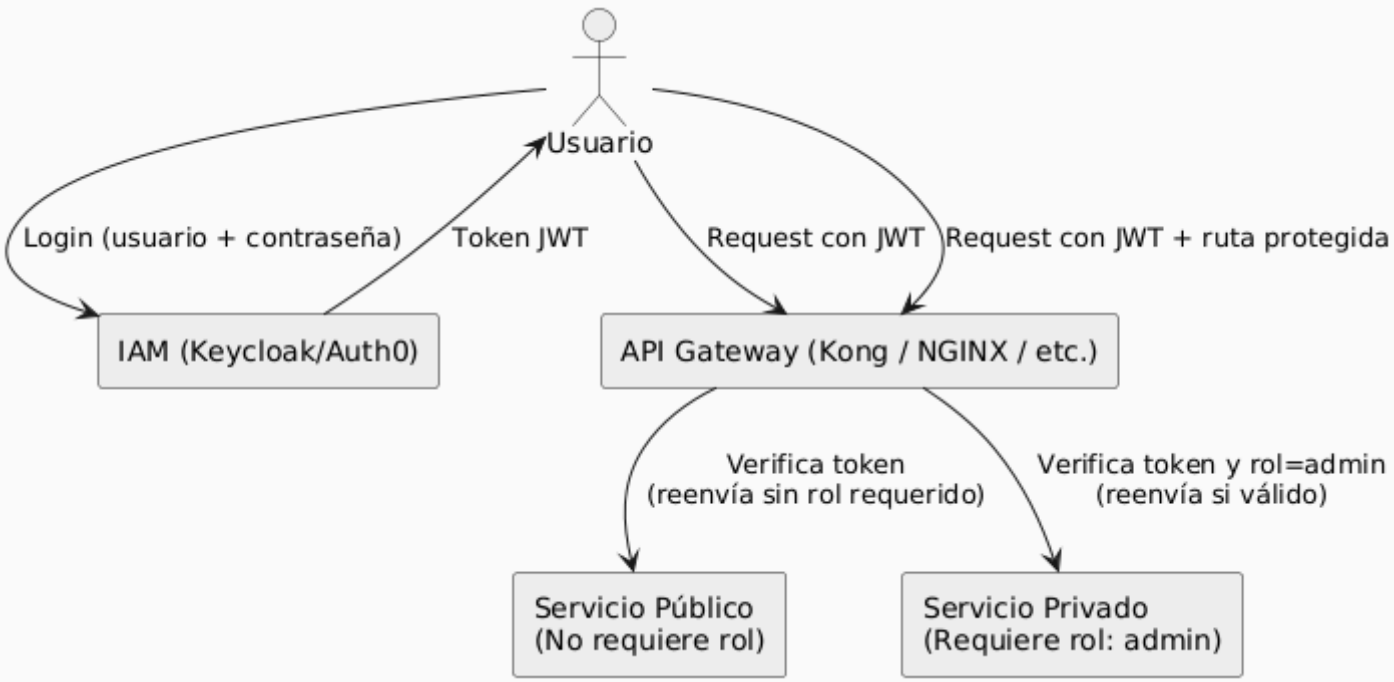
```java
@RestController
@RequestMapping("/orderDetails")
public class ApiCompositionController {

    private final RestTemplate restTemplate;

    public ApiCompositionController(RestTemplateBuilder builder) {
        this.restTemplate = builder.build();
    }

    @GetMapping("/{orderId}")
    public Map<String, Object> getOrderDetails(@PathVariable String orderId) {
        // Obtener datos de la orden
        Map<String, Object> order = restTemplate.getForObject("http://localhost:7000/orders/" +
orderId, Map.class);

        // Obtener datos del usuario
        String userId = (String) order.get("userId");
        Map<String, Object> user = restTemplate.getForObject("http://localhost:7000/users/" + userId,
Map.class);

        // Obtener detalles de los productos
        List<String> productIds = (List<String>) order.get("productIds");
        List<Map<String, Object>> products = productIds.stream()
                .map(productId -> restTemplate.getForObject("http://localhost:7000/products/" +
productId, Map.class))
                .collect(Collectors.toList());

        // Componer la respuesta
        return Map.of(
            "orderId", order.get("id"),
            "orderDate", order.get("date"),
            "total", order.get("total"),
            "customer", user,
            "products", products
        );
    }
```

Autenticación, Autorización y Seguridad en Microservicios

¿Qué pasa si cada microservicio valida usuarios, contraseñas y permisos por separado?

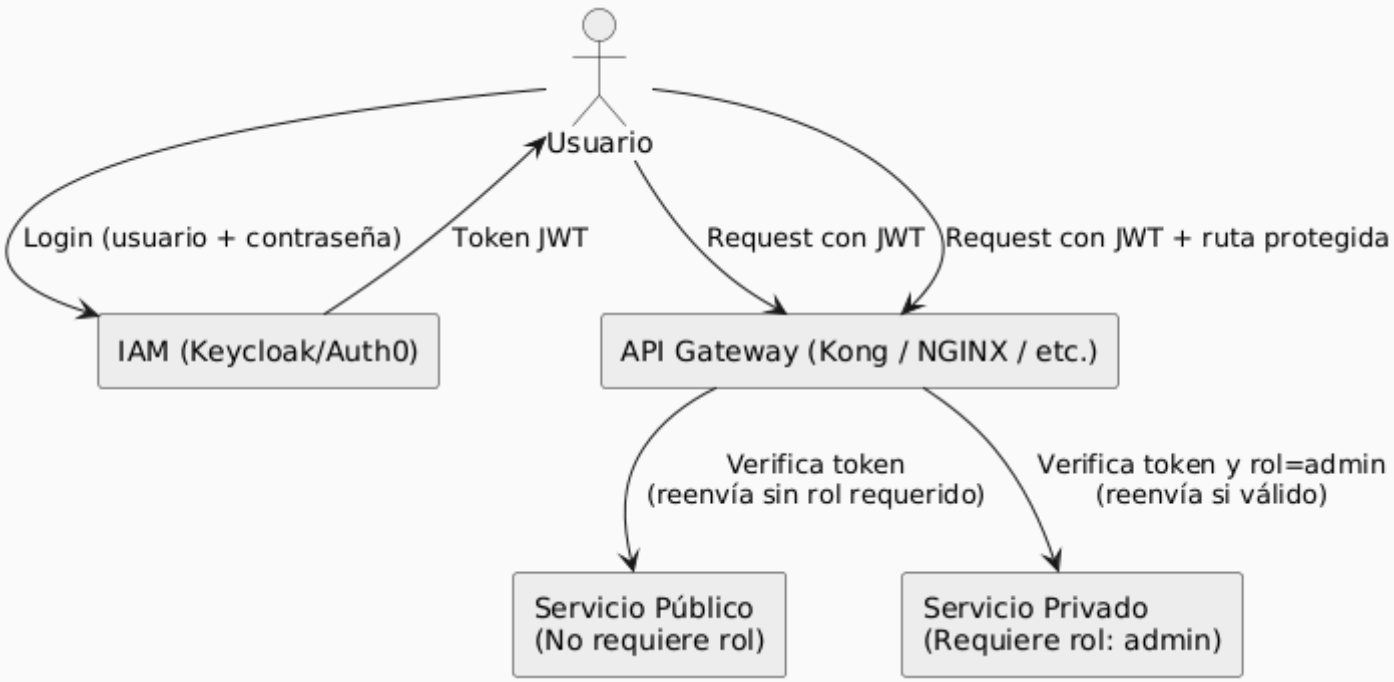
¿Por qué creen que comunmente el token se entrega al cliente y no a cada microservicio directamente?

- Eficiencia.
- Desacoplamiento.
- Seguridad.
- Delegación.



¿Por qué creen que comunmente el token se entrega al cliente y no a cada microservicio directamente?

- Eficiencia.
- Desacoplamiento.
- Seguridad.
- Delegación.

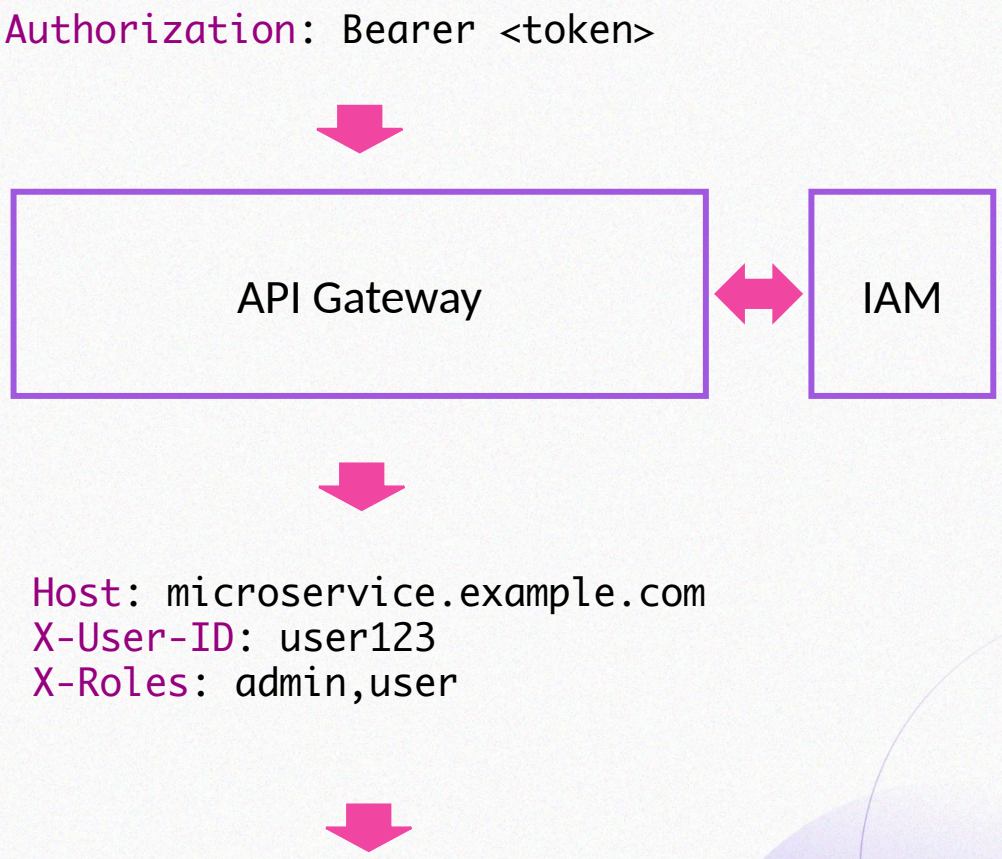


Rol del API Gateway en Identity Management

El API Gateway puede encargarse de la autenticación y autorización de usuarios al actuar como intermediario entre los clientes y los microservicios. Esto implica:

Gestión de Identidad (Identity Management):

- Delegar la autenticación a un **Identity Provider** (como Auth0, Keycloak, AWS Cognito, u OpenID Connect).
- Validar tokens de acceso (como JWT) en cada solicitud.
- Determinar los permisos del usuario (roles o claims) y asegurarse de que tengan acceso a los recursos solicitados.



Flujo típico:

- El cliente solicita autenticarse con el API Gateway.
- El Gateway redirige al Identity Provider para la autenticación.
- Una vez autenticado, el Identity Provider emite un token (por ejemplo, un JWT).
- El cliente usa este token en las solicitudes subsiguientes al API Gateway.
- El API Gateway valida el token y lo reenvía junto con la solicitud al microservicio.

CODIGO
powered by Tecsup/>

Auth0

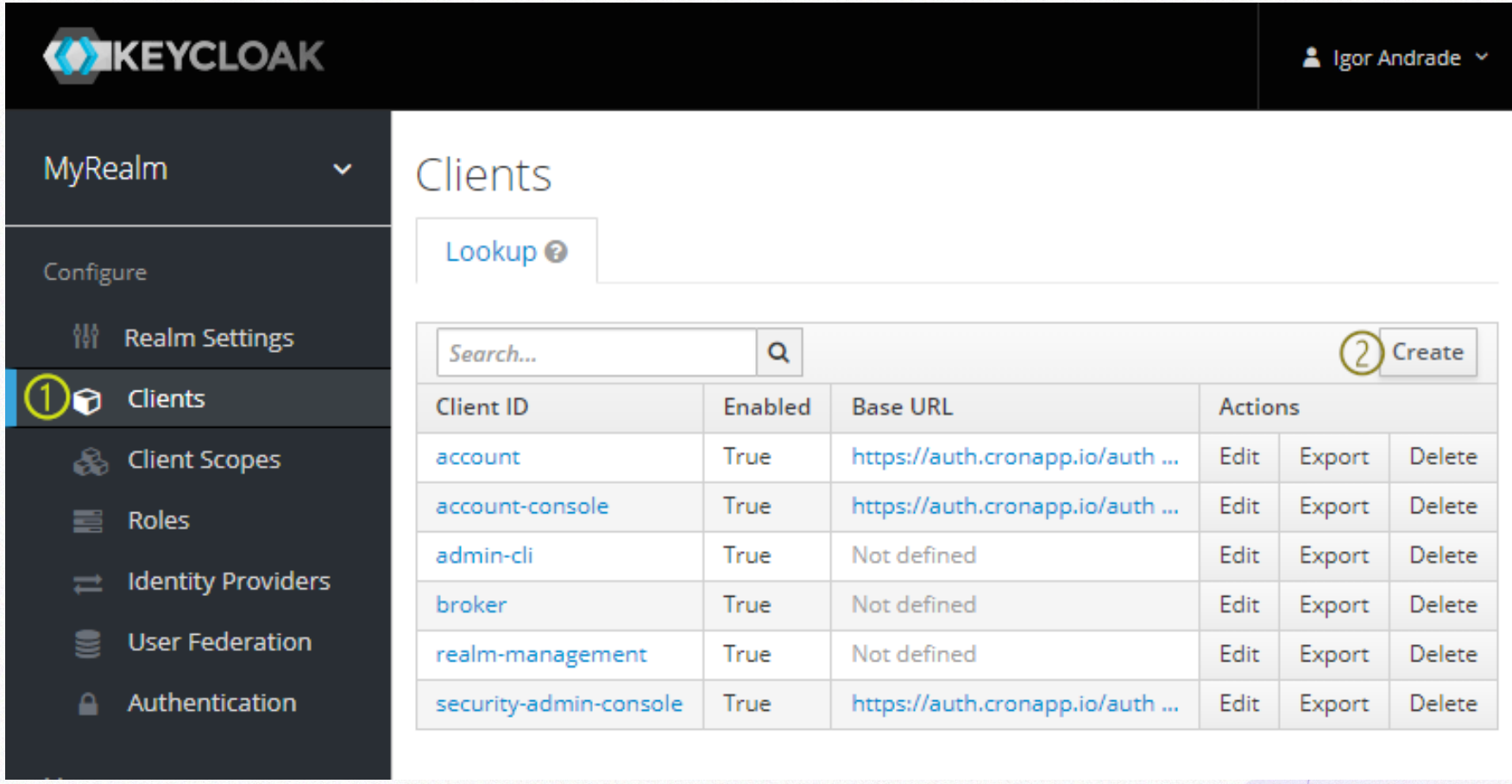
Amazon Cognito

KEYCLOAK

Keycloak es una plataforma open-source para manejar:

- Autenticación (¿quién eres?)
- Autorización (¿qué puedes hacer?)
- Gestión de usuarios, roles, sesiones y tokens (OIDC y OAuth2)

Ideal para integrarlo con microservicios, apps móviles, webapps, y más.



KeyCloak

Componente	Descripción breve
Realm	Un espacio de usuarios y configuraciones (multicliente)
Client	La aplicación que se conecta a Keycloak
User	El usuario final
Role	Permisos o capacidades del usuario
Token JWT	Prueba firmada de que el usuario está autenticado

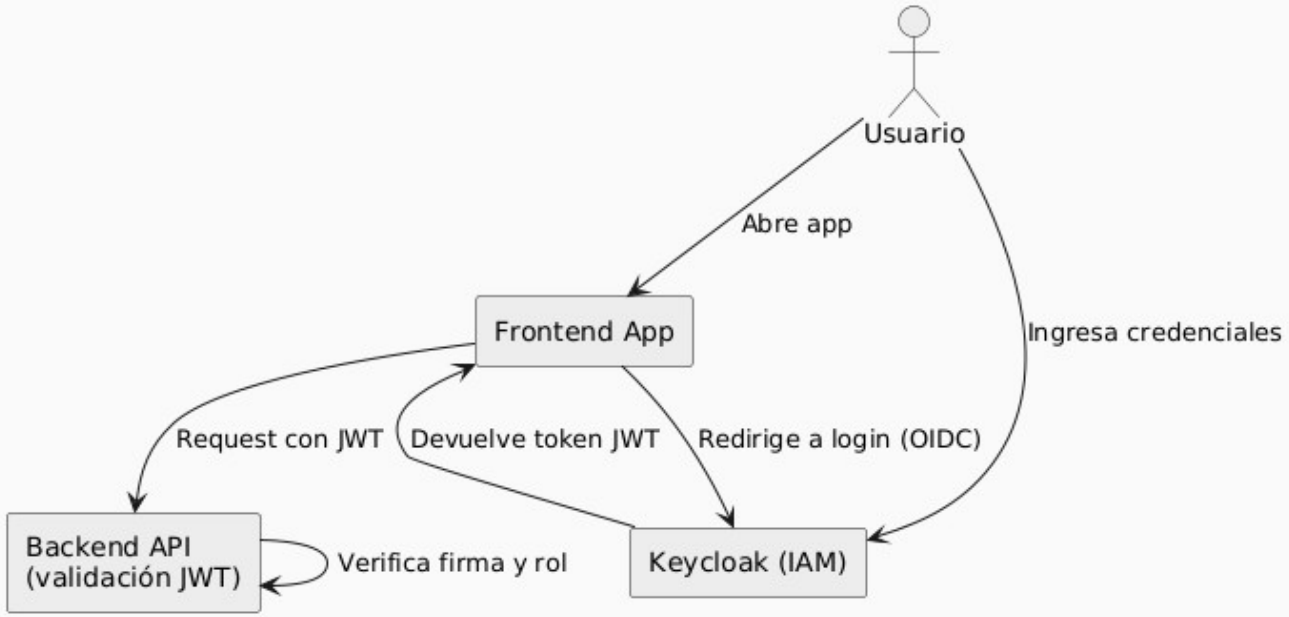
¿Cuándo usar Keycloak?

- Cuando quieres **delegar autenticación** y no reescribir lógica de login.
- Cuando tienes muchos servicios y quieres un **único sistema de usuarios y roles**.
- Cuando quieres integrar con **LDAP, Google, Facebook**, etc.
- Cuando implementas **SSO (Single Sign-On)**.

1. **Crear un Realm:**
   - Un “Realm” agrupa usuarios y aplicaciones que comparten configuraciones.
   - Navega a **Realms > Add Realm** y crea un nuevo realm, por ejemplo, `myrealm`.
2. **Crear un Cliente:**
   - En el nuevo Realm, navega a **Clients > Create**.
   - Configura el cliente:
   - **Client ID:** `api-client`
   - **Client Protocol:** `openid-connect`
- **Access Type:** `confidential`
  - Guarda los cambios y genera las credenciales del cliente (Client Secret).
3. **Crear Roles:**
   - Navega a **Roles** y crea roles como `user` o `admin`.
4. **Crear Usuarios:**
   - Ve a **Users > Add User**, crea un usuario, y asigna roles.

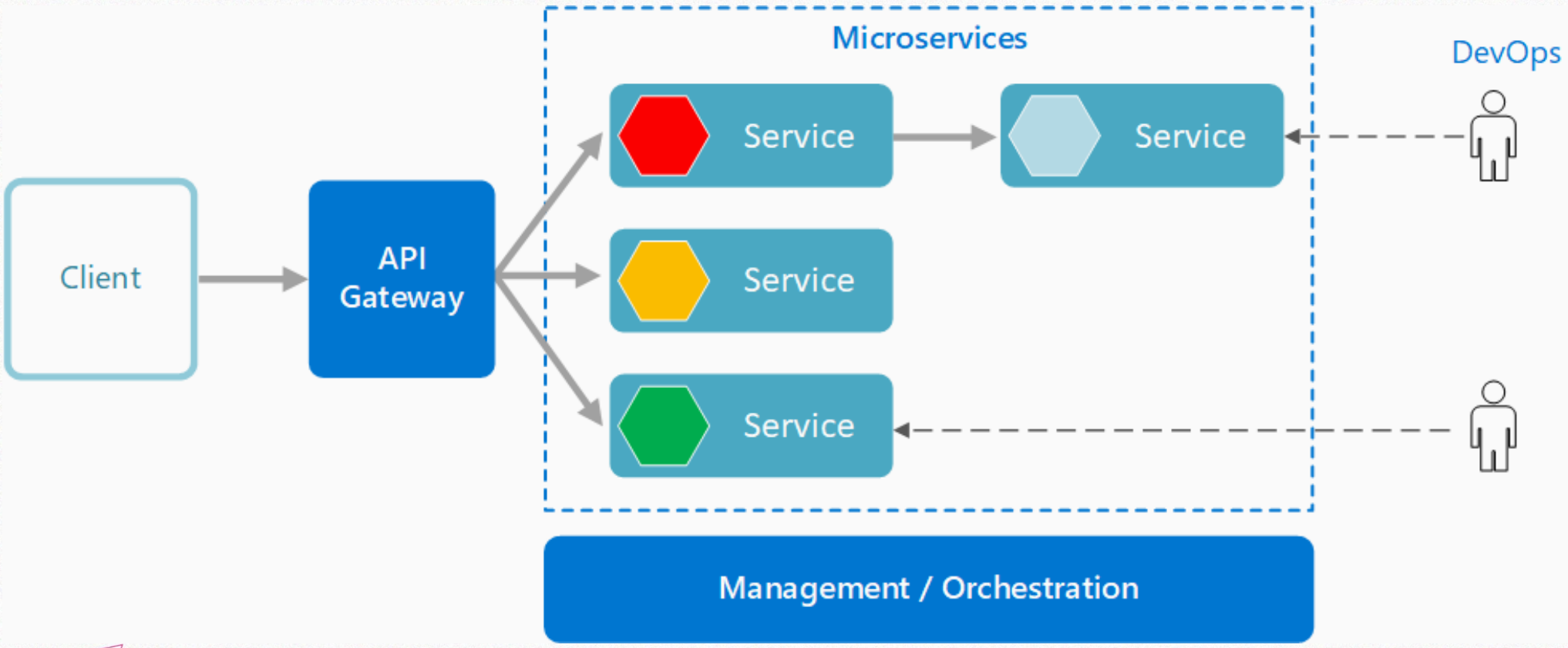
KeyCloak

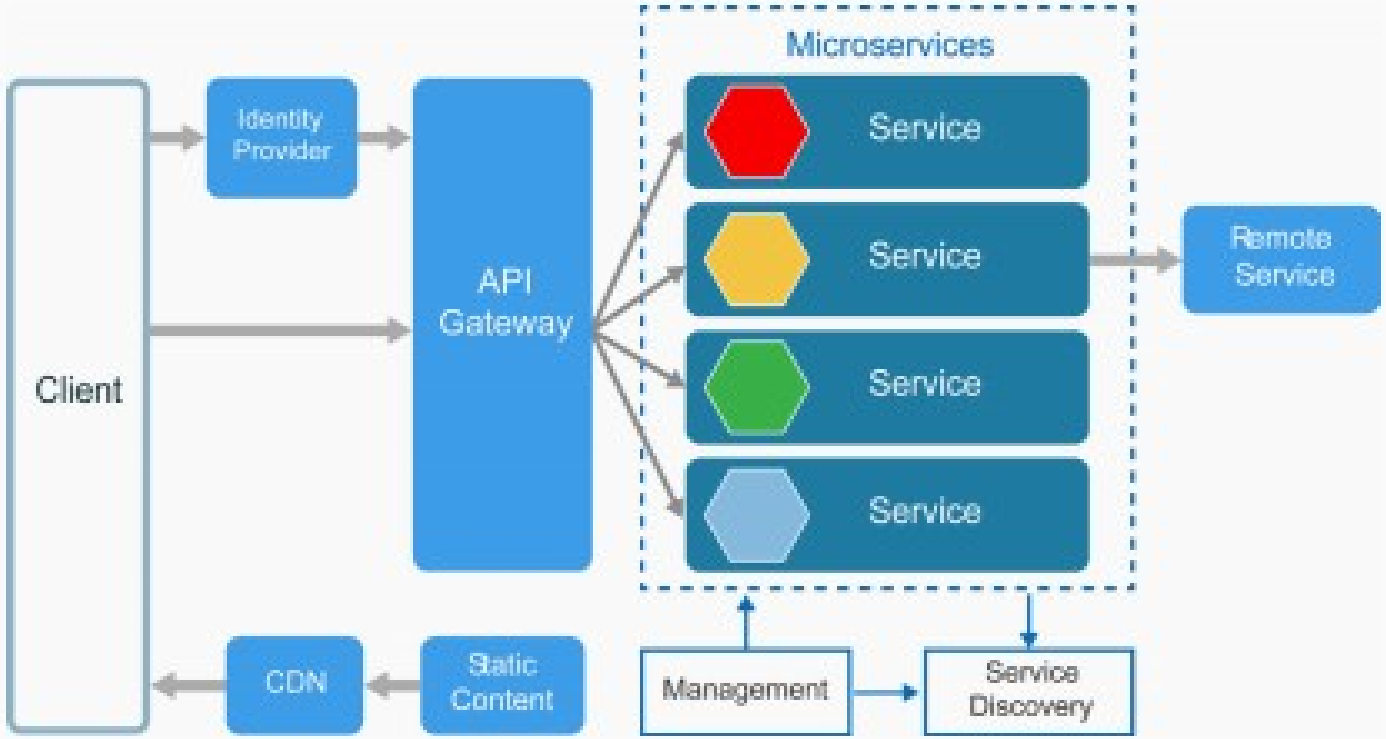
Componente	Descripción breve
Realm	Un espacio de usuarios y configuraciones (multicliente)
Client	La aplicación que se conecta a Keycloak
User	El usuario final
Role	Permisos o capacidades del usuario
Token JWT	Prueba firmada de que el usuario está autenticado



¿Cuándo usar Keycloak?

- Cuando quieres **delegar autenticación** y no reescribir lógica de login.
- Cuando tienes muchos servicios y quieres un **único sistema de usuarios y roles**.
- Cuando quieres integrar con **LDAP, Google, Facebook**, etc.
- Cuando implementas **SSO (Single Sign-On)**.


Microservices
DevOps
Client
API Gateway
Service
Service
Service
Service
Management / Orchestration


Microservices
Client
Identity Provider
API Gateway
Service
Service
Service
Service
Remote Service
CDN
Static Content
Management
Service Discovery

Tolerancia a Fallos

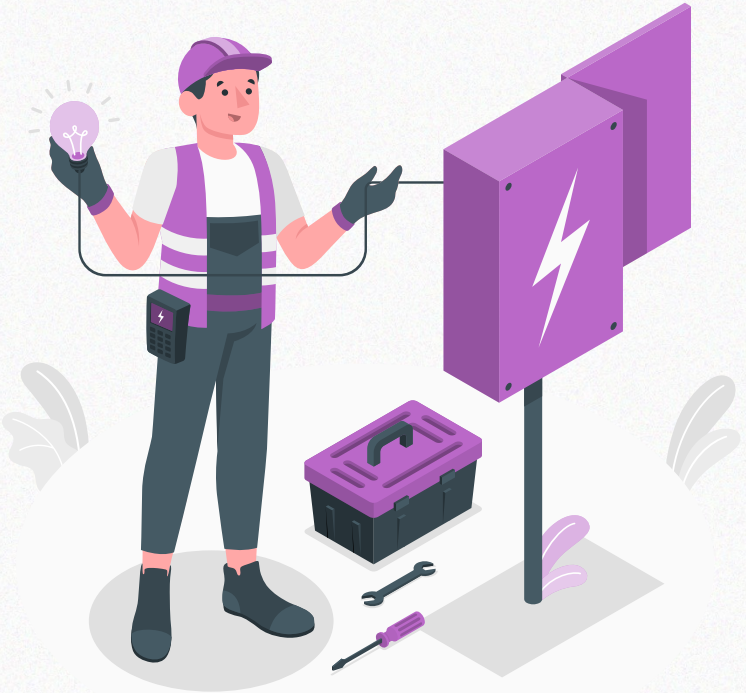
¿Qué pasaría si un microservicio lento comienza a colapsar… y aún así todos los demás siguen llamándolo sin descanso?

Circuit Breaker

El concepto se inspira en los interruptores eléctricos que cortan el suministro cuando detectan un sobrevoltaje. En software, un **Circuit Breaker** actúa como un guardián que supervisa las llamadas a servicios externos o componentes y, si detecta un problema continuo, “abre el circuito” para bloquear temporalmente las solicitudes y proteger el sistema.

Un **Circuit Breaker** (cortacircuito) es un componente que **detiene temporalmente el flujo de solicitudes** a un servicio que está fallando, para evitar saturar el sistema y dar tiempo a que se recupere.

Estado	Significado	Acción del sistema
Closed	Todo funciona bien	Las llamadas pasan
Open	Se detectaron fallos consecutivos	Se bloquean llamadas (fallo inmediato)
Half-Open	Se permite una llamada de prueba	Si tiene éxito, vuelve a Closed



Circuit Breaker

El Circuit Breaker tiene tres estados principales:

1. **Cerrado (Closed):**

• Es el estado normal.

• Las solicitudes pasan libremente hacia el recurso externo.

• El Circuit Breaker supervisa las respuestas. Si detecta demasiados errores (por ejemplo, un porcentaje alto de fallos), cambia a “Abierto”.

2. **Abierto (Open):**

• Todas las solicitudes se bloquean inmediatamente, sin llegar al recurso externo.

• Esto evita sobrecargar un sistema que ya está fallando.

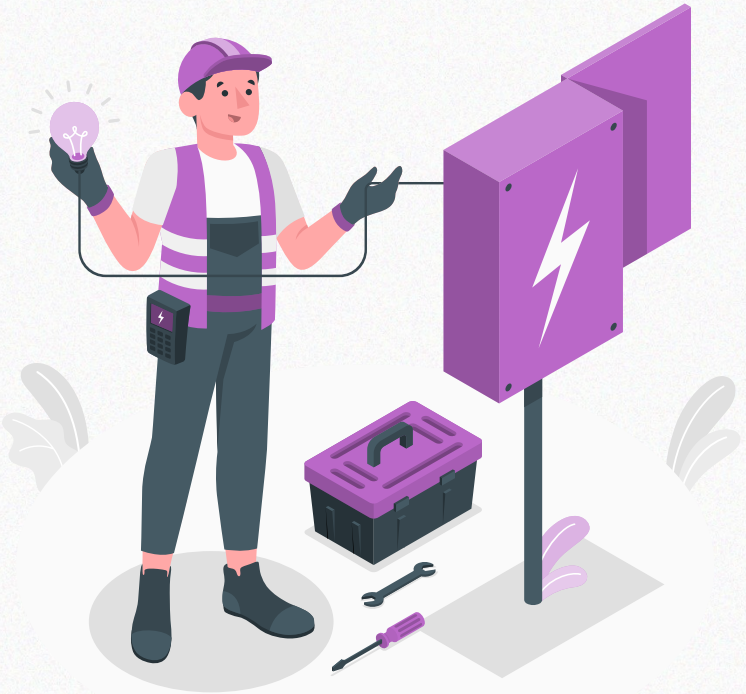
• Permanece en este estado durante un tiempo definido (timeout o ventana de enfriamiento).

3. **Semiabierto (Half-Open):**

• Después del período de timeout, el Circuit Breaker permite que un número limitado de solicitudes pasen al recurso externo como prueba.

• Si las solicitudes tienen éxito, el Circuit Breaker vuelve a “Cerrado”.

• Si las solicitudes fallan, regresa al estado “Abierto”.



¿Cómo funciona el Circuit Breaker?

Flujo básico:

1. Una solicitud llega al Circuit Breaker.

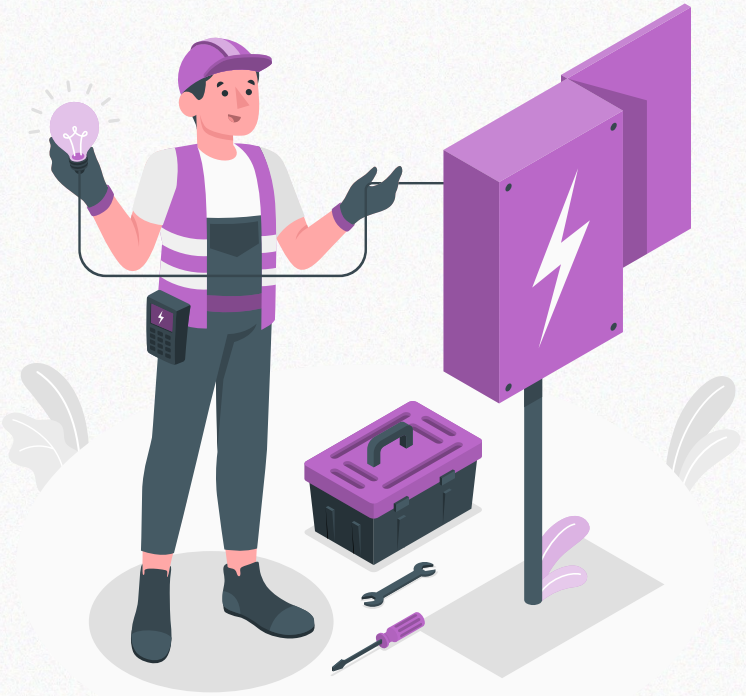
2. El Circuit Breaker verifica su estado:

• Si está en “Cerrado”, envía la solicitud al recurso externo y supervisa el resultado.

• Si está en “Abierto”, rechaza la solicitud inmediatamente y devuelve un error o una respuesta predeterminada.

• Si está en “Semiabierto”, permite que algunas solicitudes pasen para probar el estado del recurso externo.

3. El Circuit Breaker registra el éxito o fallo de las solicitudes para actualizar su estado.



¿Cómo funciona el Circuit Breaker?

```
CircuitBreaker circuitBreaker = CircuitBreaker.ofDefaults("pagoService");

Supplier<String> resultado = CircuitBreaker.decorateSupplier(circuitBreaker, () -> {
    return pagoService.realizarPago();
});

Try<String> intento = Try.ofSupplier(resultado)
    .recover(throwable -> "Servicio no disponible");
```

```javascript
const CircuitBreaker = require('opossum');

// Función que hace una solicitud a un servicio externo
function requestService() {
    return new Promise((resolve, reject) => {
        // Simula un servicio externo que a veces falla
        const isSuccess = Math.random() > 0.7; // 30% de éxito
        if (isSuccess) {
            resolve("Servicio exitoso");
        } else {
            reject("Error en el servicio");
        }
    });
}

// Configuración del Circuit Breaker
const breaker = new CircuitBreaker(requestService, {
    timeout: 5000, // Tiempo máximo para una solicitud
    errorThresholdPercentage: 50, // Cambia a "Abierto" si más del 50% fallan
    resetTimeout: 10000 // Tiempo para probar nuevamente
});

// Manejo de eventos
breaker.on('open', () => console.log('Circuito abierto: demasiados errores'));
breaker.on('halfOpen', () => console.log('Circuito semiabierto: probando el servicio'));
breaker.on('close', () => console.log('Circuito cerrado: el servicio se recuperó'));

// Llamadas al servicio
setInterval(() => {
    breaker.fire()
        .then(console.log)
        .catch(console.error);
}, 1000);
```

Beneficios del Circuit Breaker

1. **Evita cascadas de fallos:**

• Si un servicio está fallando, el Circuit Breaker evita que los demás servicios hagan llamadas inútiles que podrían sobrecargar el sistema.

2. **Protección del rendimiento:**

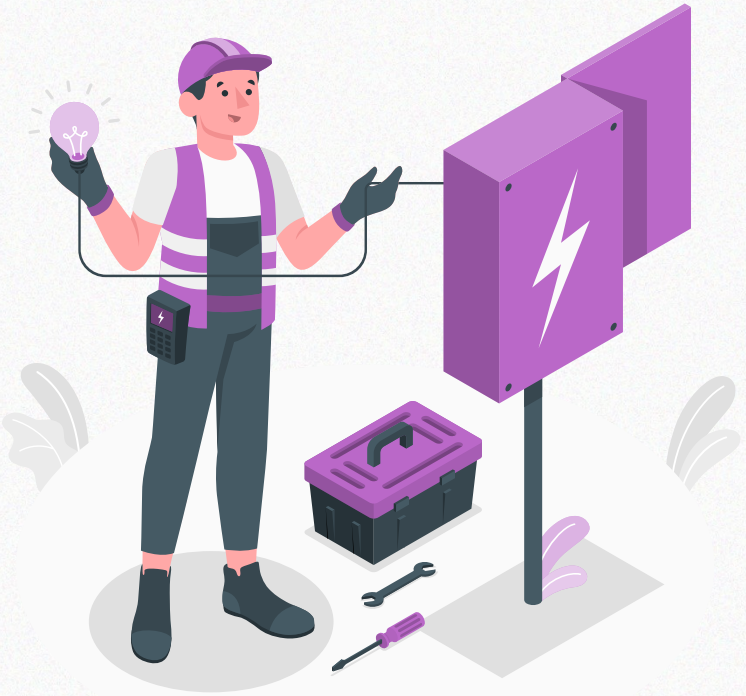
• Al reducir el número de llamadas fallidas, se evita el desperdicio de recursos en tiempo de espera y reintentos.

3. **Recuperación automática:**

• Permite al sistema recuperar gradualmente la conexión con el servicio fallido (estado “Semiabierto”).

4. **Respuesta rápida:**

• Cuando el Circuit Breaker está “Abierto”, las solicitudes fallan inmediatamente, mejorando la experiencia del usuario al no tener que esperar tiempos de espera largos.



```java
import com.netflix.hystrix.HystrixCommand;
import com.netflix.hystrix.HystrixCommandGroupKey;

public class MyServiceCommand extends HystrixCommand<String> {
    public MyServiceCommand() {
        super(HystrixCommandGroupKey.Factory.asKey("MyServiceGroup"));
    }

    @Override
    protected String run() throws Exception {
        // Lógica para llamar al servicio externo
        if (Math.random() > 0.7) {
            return "Servicio exitoso";
        } else {
            throw new RuntimeException("Error en el servicio");
        }
    }

    @Override
    protected String getFallback() {
        return "Respuesta predeterminada";
    }
}

// Uso del Circuit Breaker
MyServiceCommand command = new MyServiceCommand();
System.out.println(command.execute());
```

Retries con Backoff Exponencial

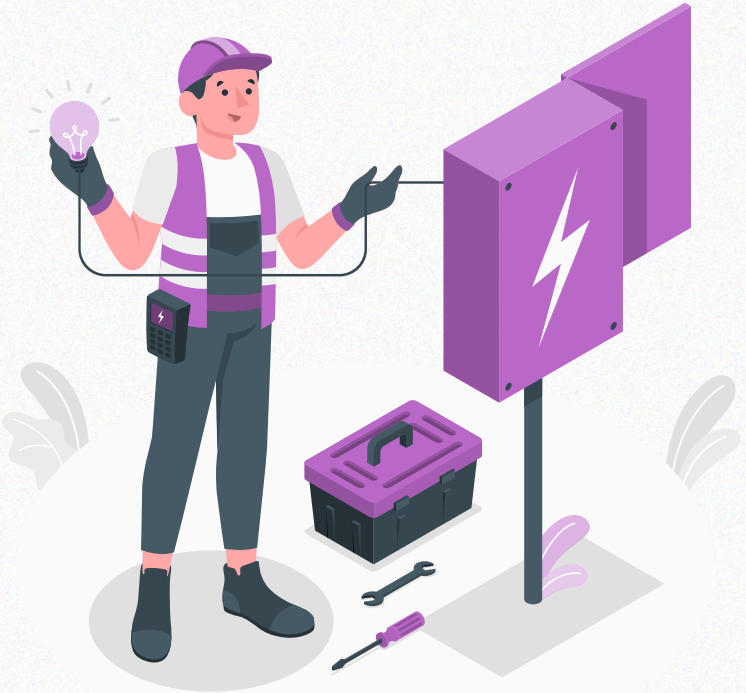
Cuando una solicitud falla debido a problemas temporales (como errores de red), se vuelve a intentar después de un intervalo. El intervalo aumenta exponencialmente después de cada intento fallido.

Beneficios:

- Útil para errores transitorios.
- Reduce la presión en el sistema fallido al espaciar los intentos.

Implementación:

- Backoff exponencial: El tiempo entre intentos crece exponencialmente (por ejemplo, 1s, 2s, 4s, 8s…).
- Jitter: Se agrega una variación aleatoria para evitar picos de tráfico cuando muchos clientes realizan reintentos.



Retries con Backoff Exponencial

```
function retryWithBackoff(fn, retries = 5, delay = 1000) {
    return new Promise((resolve, reject) => {
        fn()
            .then(resolve)
            .catch(err => {
                if (retries > 0) {
                    const newDelay = delay * 2;
                    setTimeout(() => {
                        retryWithBackoff(fn, retries - 1, newDelay).then(resolve).catch(reject);
                    }, newDelay);
                } else {
                    reject(err);
                }
            });
    });
}

// Uso:
retryWithBackoff(() => fetch('https://example.com'))
    .then(response => console.log('Éxito:', response))
    .catch(err => console.error('Fallo:', err));
```

Consultas en Microservicios

CQRS (Command Query Responsibility Segregation)

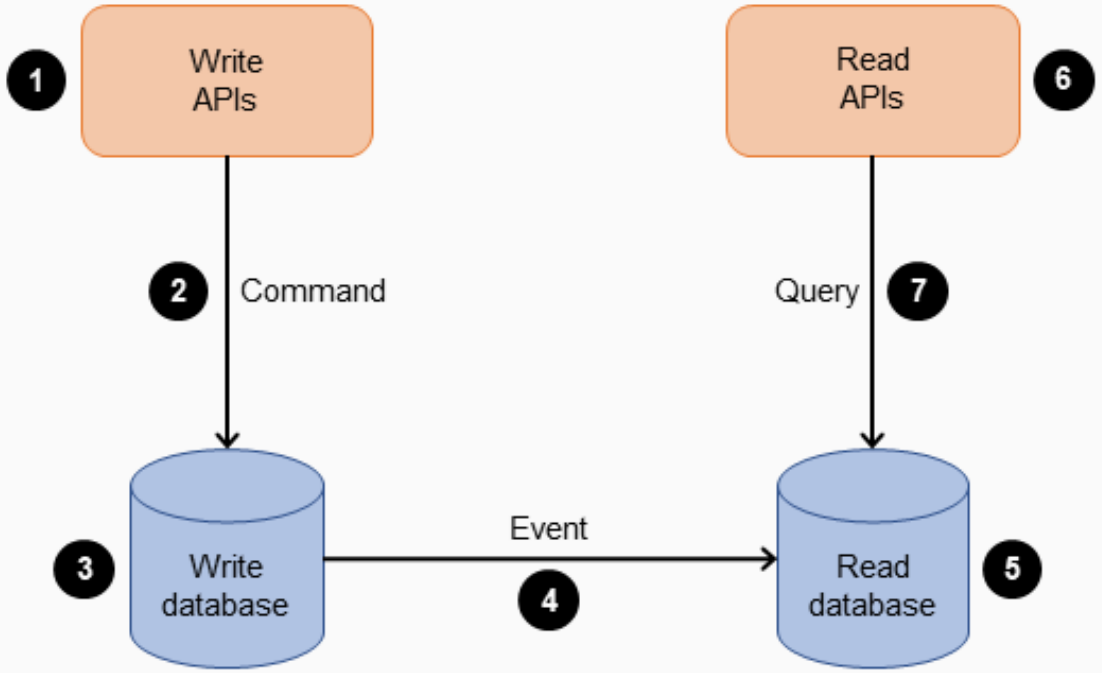
CQRS separa las responsabilidades de:

1. **Escribir datos**: Operaciones que modifican el estado del sistema, como crear, actualizar o eliminar datos.

2. **Leer datos**: Operaciones que consultan el estado del sistema.

En lugar de usar el mismo modelo de datos y lógica para ambas operaciones, CQRS define **dos modelos separados**:

- Un **modelo de escritura** para manejar comandos.
- Un **modelo de lectura** optimizado para consultas.



¿Por qué separar lectura y escritura?

En sistemas tradicionales (CRUD), un solo modelo y base de datos gestionan tanto lecturas como escrituras. Esto puede causar:

1. **Conflictos de diseño**:

• Las lecturas requieren índices, desnormalización y optimización para consultas rápidas.

• Las escrituras requieren normalización, integridad referencial y transacciones.

2. **Bajo rendimiento**:

• Las consultas pesadas pueden ralentizar las operaciones de escritura.

3. **Escalabilidad limitada**:

• Es difícil escalar operaciones de lectura y escritura de forma independiente.

Componentes clave de CQRS

1. **Command Handlers:**

• Reciben comandos (por ejemplo, “crear usuario”, “actualizar pedido”).

• Realizan validaciones y aplican lógica de negocio.

• Pueden publicar eventos si se utiliza Event Sourcing.

2. **Query Handlers:**

• Ejecutan consultas basadas en parámetros del cliente.

• Devuelven datos optimizados para el consumo (por ejemplo, vistas planas o agregadas).

3. **Eventos (opcional):**

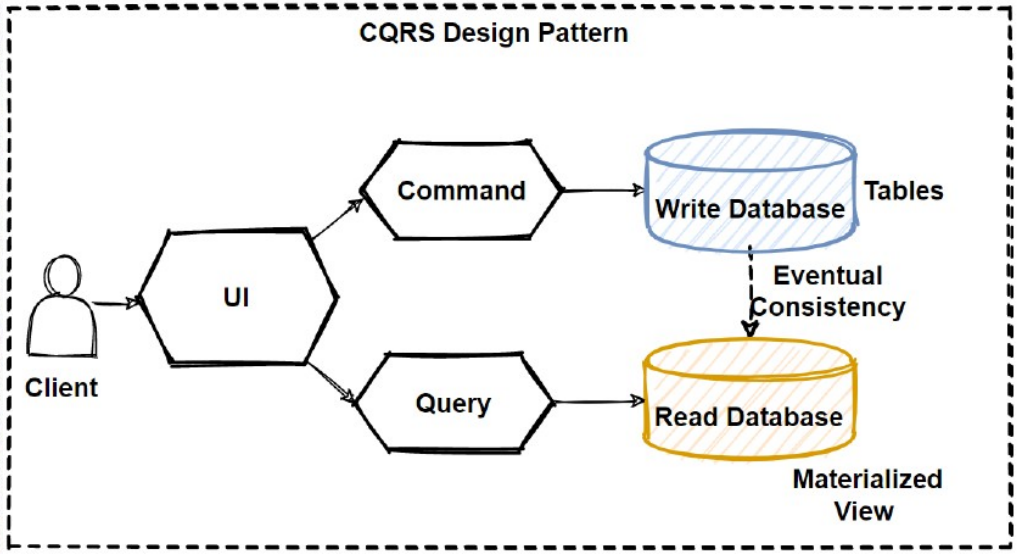
• Cuando se combinan con Event Sourcing, los cambios de estado se almacenan como eventos inmutables.

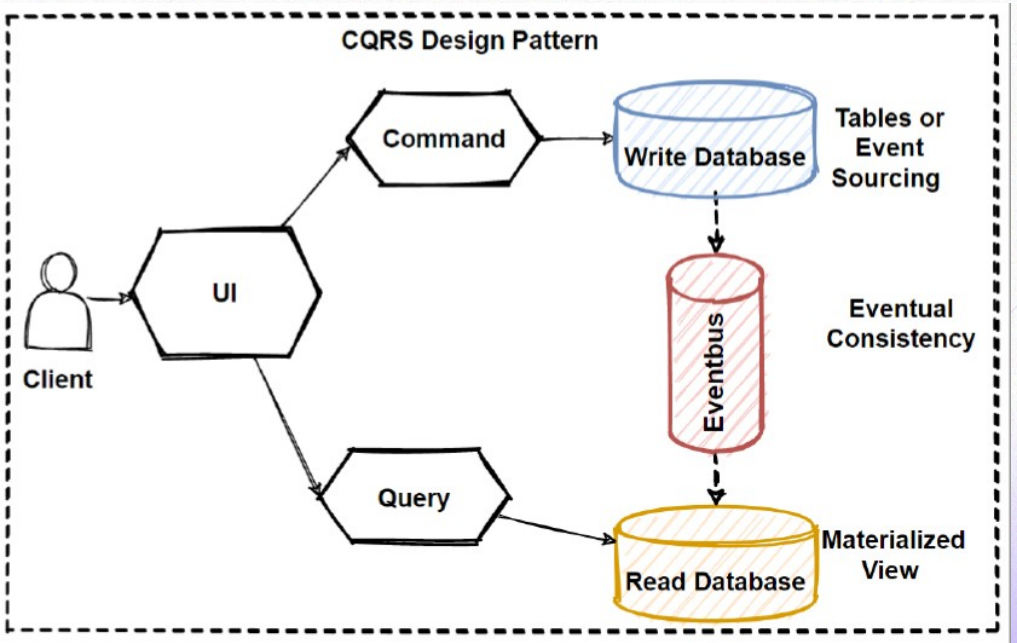
• Los eventos pueden ser procesados para construir modelos de consulta.

4. **Bases de datos separadas:**

• Una base de datos para escribir (puede ser transaccional y normalizada).

• Una o más bases de datos para leer (pueden ser denormalizadas y optimizadas para consultas).

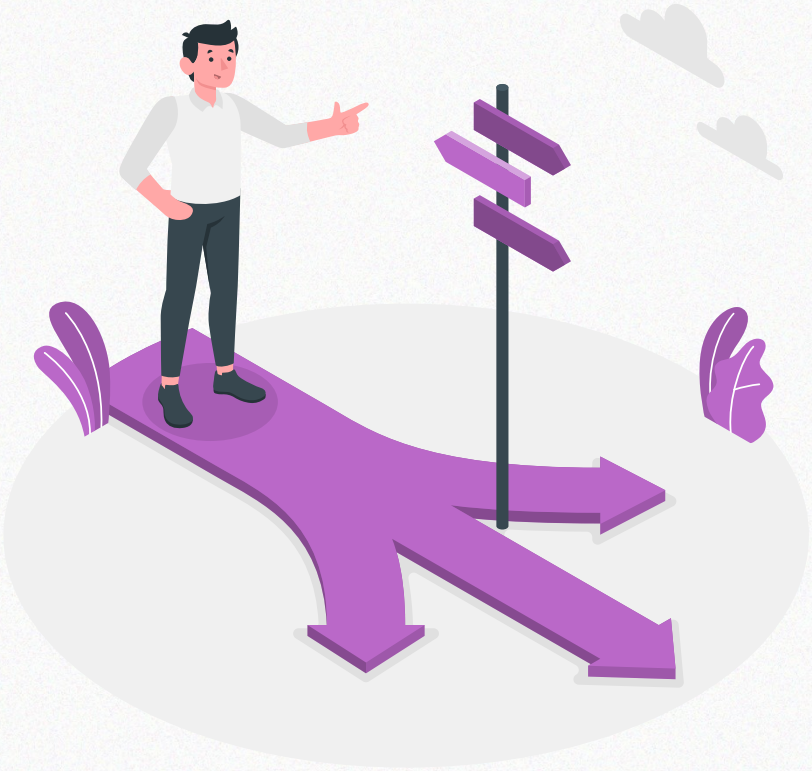




¿Cuándo usar CQRS?

CQRS no es necesario para todas las aplicaciones. Es más adecuado para sistemas con las siguientes características:

- Requisitos de lectura y escritura muy diferentes.
- Escenarios con alto volumen de lecturas o escrituras.
- Lógica de negocio compleja que justifica separar la escritura de las consultas.
- Necesidad de escalabilidad extrema.
- Uso de Event Sourcing para rastrear cambios en el estado del sistema.



Ejemplo práctico de CQRS

Microservicio de pedidos:

•**Comandos:** Crear pedido, agregar producto, confirmar pedido, cancelar pedido.
•**Consultas:** Obtener detalles del pedido, listar pedidos por cliente, obtener el estado del pedido.
•**Modelo de escritura:** Tabla "Pedidos" con información detallada del pedido, productos, cliente, etc.
•**Modelo de lectura:** Vista materializada con información resumida del pedido para mostrar en la interfaz de usuario.

Microservicio de catálogo:
•**Comandos:** Agregar producto, actualizar información del producto, eliminar producto.
•**Consultas:** Obtener detalles del producto, buscar productos por nombre o categoría, obtener productos relacionados.
•**Modelo de escritura:** Tabla "Productos" con información detallada de cada producto.
•**Modelo de lectura:** Índice de búsqueda optimizado para búsquedas rápidas por nombre y categoría.

Ejemplo práctico de CQRS

Modelo de escritura:

```
class Order {
    constructor(orderId, customerId, items) {
        this.orderId = orderId;
        this.customerId = customerId;
        this.items = items;
        this.status = 'Pending';
    }


    confirm() {
        if (this.status !== 'Pending') {
            throw new Error('Order cannot be confirmed');
        }
        this.status = 'Confirmed';
    }
}
```

Command handler:

```
function handleConfirmOrderCommand(command) {
    const order = orderRepository.getById(command.orderId);
    order.confirm();
    orderRepository.save(order);
}
```

Query Side:

```
{
    "orderId": "12345",
    "customerName": "John Doe",
    "totalPrice": 99.99,
    "items": [
        { "name": "Product A", "quantity": 1, "price": 50 },
        { "name": "Product B", "quantity": 1, "price": 49.99 }
    ]
}
```

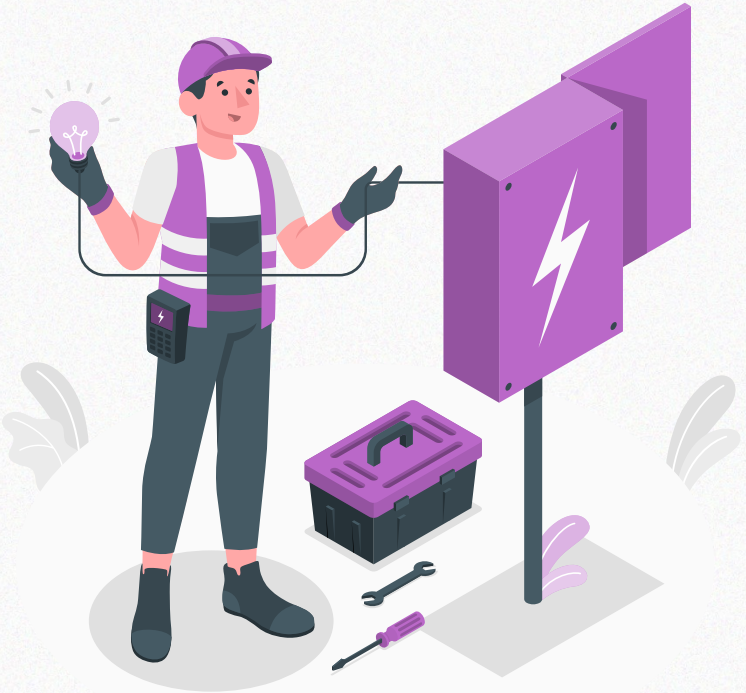
Query handler:

```
function getOrderDetails(orderId) {
    return orderReadModel.getById(orderId); // Base de datos optimizada para consultas
}
```

Tolerancia a Fallos

Circuit Breaker

El concepto se inspira en los interruptores eléctricos que cortan el suministro cuando detectan un sobrevoltaje. En software, un **Circuit Breaker** actúa como un guardián que supervisa las llamadas a servicios externos o componentes y, si detecta un problema continuo, “abre el circuito” para bloquear temporalmente las solicitudes y proteger el sistema.



Circuit Breaker

El Circuit Breaker tiene tres estados principales:

1. **Cerrado (Closed):**

• Es el estado normal.

• Las solicitudes pasan libremente hacia el recurso externo.

• El Circuit Breaker supervisa las respuestas. Si detecta demasiados errores (por ejemplo, un porcentaje alto de fallos), cambia a “Abierto”.

2. **Abierto (Open):**

• Todas las solicitudes se bloquean inmediatamente, sin llegar al recurso externo.

• Esto evita sobrecargar un sistema que ya está fallando.

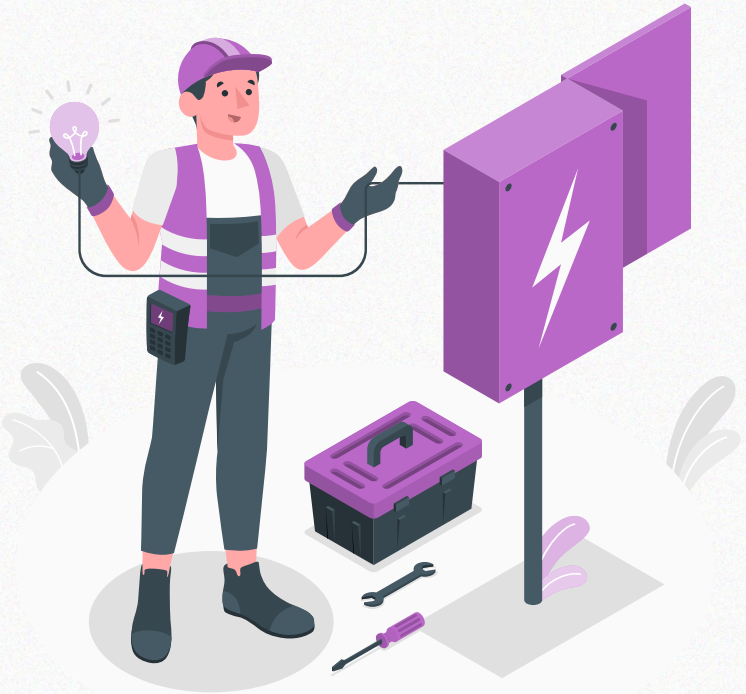
• Permanece en este estado durante un tiempo definido (timeout o ventana de enfriamiento).

3. **Semiabierto (Half-Open):**

• Después del período de timeout, el Circuit Breaker permite que un número limitado de solicitudes pasen al recurso externo como prueba.

• Si las solicitudes tienen éxito, el Circuit Breaker vuelve a “Cerrado”.

• Si las solicitudes fallan, regresa al estado “Abierto”.



¿Cómo funciona el Circuit Breaker?

Flujo básico:

1. Una solicitud llega al Circuit Breaker.

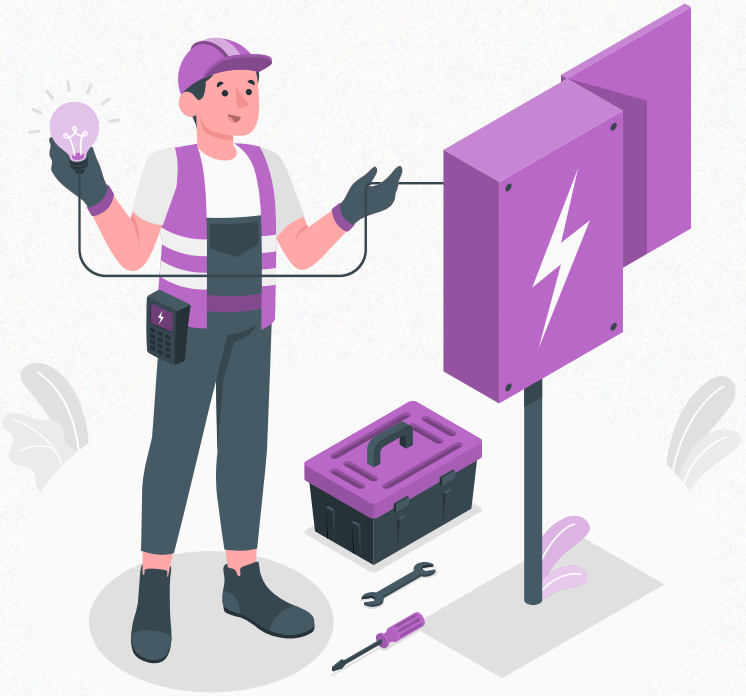
2. El Circuit Breaker verifica su estado:

• Si está en “Cerrado”, envía la solicitud al recurso externo y supervisa el resultado.

• Si está en “Abierto”, rechaza la solicitud inmediatamente y devuelve un error o una respuesta predeterminada.

• Si está en “Semiabierto”, permite que algunas solicitudes pasen para probar el estado del recurso externo.

3. El Circuit Breaker registra el éxito o fallo de las solicitudes para actualizar su estado.



Beneficios del Circuit Breaker

1. **Evita cascadas de fallos:**

• Si un servicio está fallando, el Circuit Breaker evita que los demás servicios hagan llamadas inútiles que podrían sobrecargar el sistema.

2. **Protección del rendimiento:**

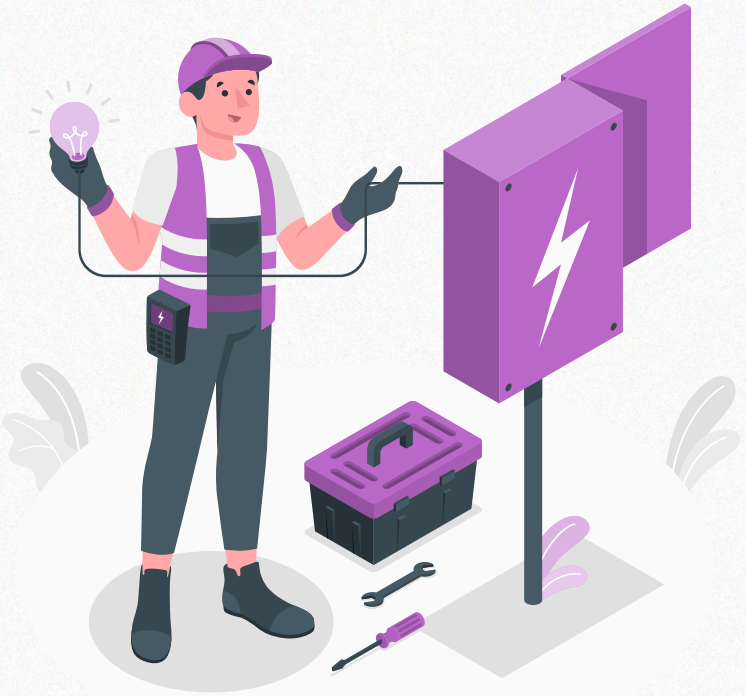
• Al reducir el número de llamadas fallidas, se evita el desperdicio de recursos en tiempo de espera y reintentos.

3. **Recuperación automática:**

• Permite al sistema recuperar gradualmente la conexión con el servicio fallido (estado “Semiabierto”).

4. **Respuesta rápida:**

• Cuando el Circuit Breaker está “Abierto”, las solicitudes fallan inmediatamente, mejorando la experiencia del usuario al no tener que esperar tiempos de espera largos.



```javascript
const CircuitBreaker = require('opossum');

// Función que hace una solicitud a un servicio externo
function requestService() {
    return new Promise((resolve, reject) => {
        // Simula un servicio externo que a veces falla
        const isSuccess = Math.random() > 0.7; // 30% de éxito
        if (isSuccess) {
            resolve("Servicio exitoso");
        } else {
            reject("Error en el servicio");
        }
    });
}

// Configuración del Circuit Breaker
const breaker = new CircuitBreaker(requestService, {
    timeout: 5000, // Tiempo máximo para una solicitud
    errorThresholdPercentage: 50, // Cambia a "Abierto" si más del 50% fallan
    resetTimeout: 10000 // Tiempo para probar nuevamente
});

// Manejo de eventos
breaker.on('open', () => console.log('Circuito abierto: demasiados errores'));
breaker.on('halfOpen', () => console.log('Circuito semiabierto: probando el servicio'));
breaker.on('close', () => console.log('Circuito cerrado: el servicio se recuperó'));

// Llamadas al servicio
setInterval(() => {
    breaker.fire()
        .then(console.log)
        .catch(console.error);
}, 1000);
```

```java
import com.netflix.hystrix.HystrixCommand;
import com.netflix.hystrix.HystrixCommandGroupKey;

public class MyServiceCommand extends HystrixCommand<String> {
    public MyServiceCommand() {
        super(HystrixCommandGroupKey.Factory.asKey("MyServiceGroup"));
    }

    @Override
    protected String run() throws Exception {
        // Lógica para llamar al servicio externo
        if (Math.random() > 0.7) {
            return "Servicio exitoso";
        } else {
            throw new RuntimeException("Error en el servicio");
        }
    }

    @Override
    protected String getFallback() {
        return "Respuesta predeterminada";
    }
}

// Uso del Circuit Breaker
MyServiceCommand command = new MyServiceCommand();
System.out.println(command.execute());
```

Retries con Backoff Exponencial

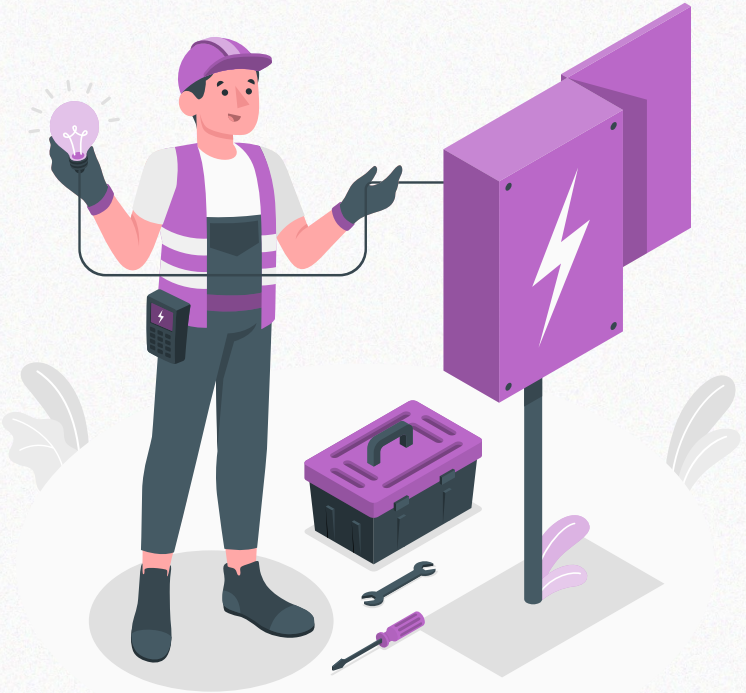
Cuando una solicitud falla debido a problemas temporales (como errores de red), se vuelve a intentar después de un intervalo. El intervalo aumenta exponencialmente después de cada intento fallido.

Beneficios:

- Útil para errores transitorios.
- Reduce la presión en el sistema fallido al espaciar los intentos.

Implementación:

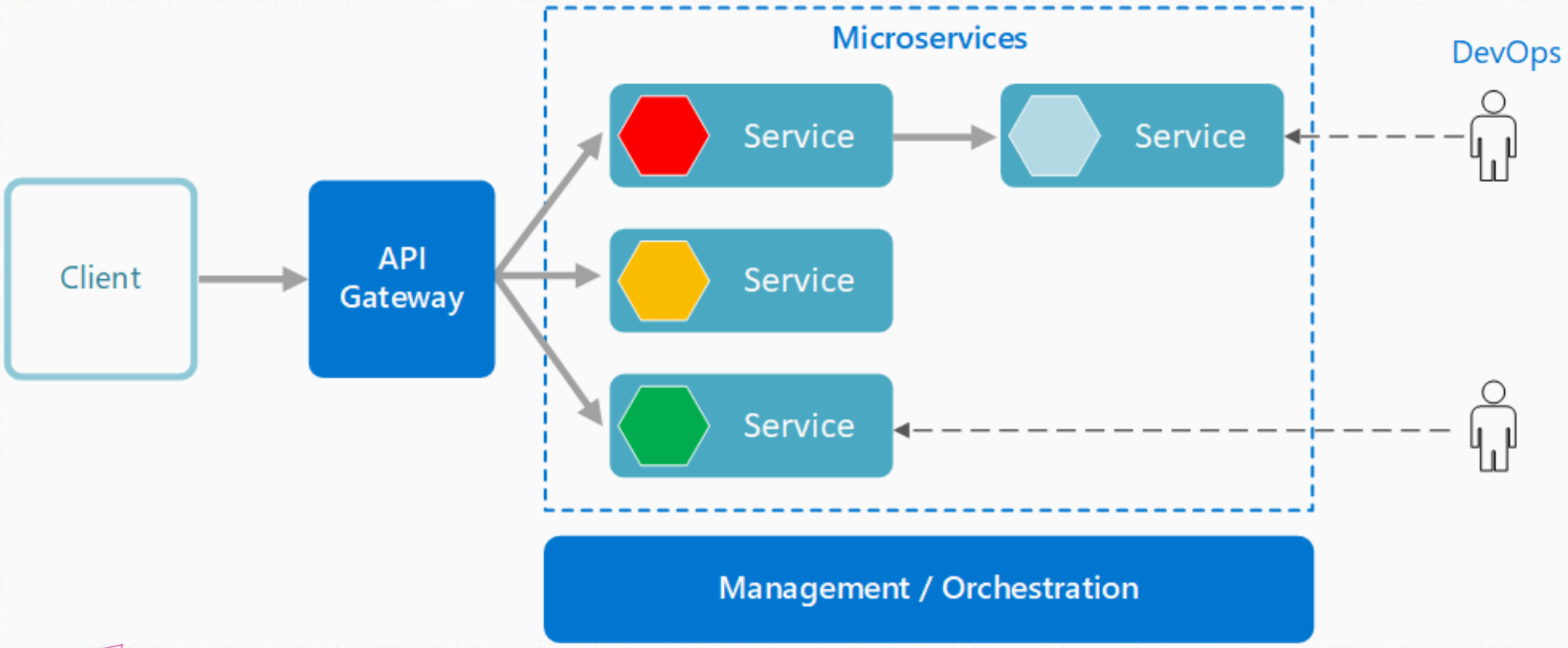
- Backoff exponencial: El tiempo entre intentos crece exponencialmente (por ejemplo, 1s, 2s, 4s, 8s…).
- Jitter: Se agrega una variación aleatoria para evitar picos de tráfico cuando muchos clientes realizan reintentos.

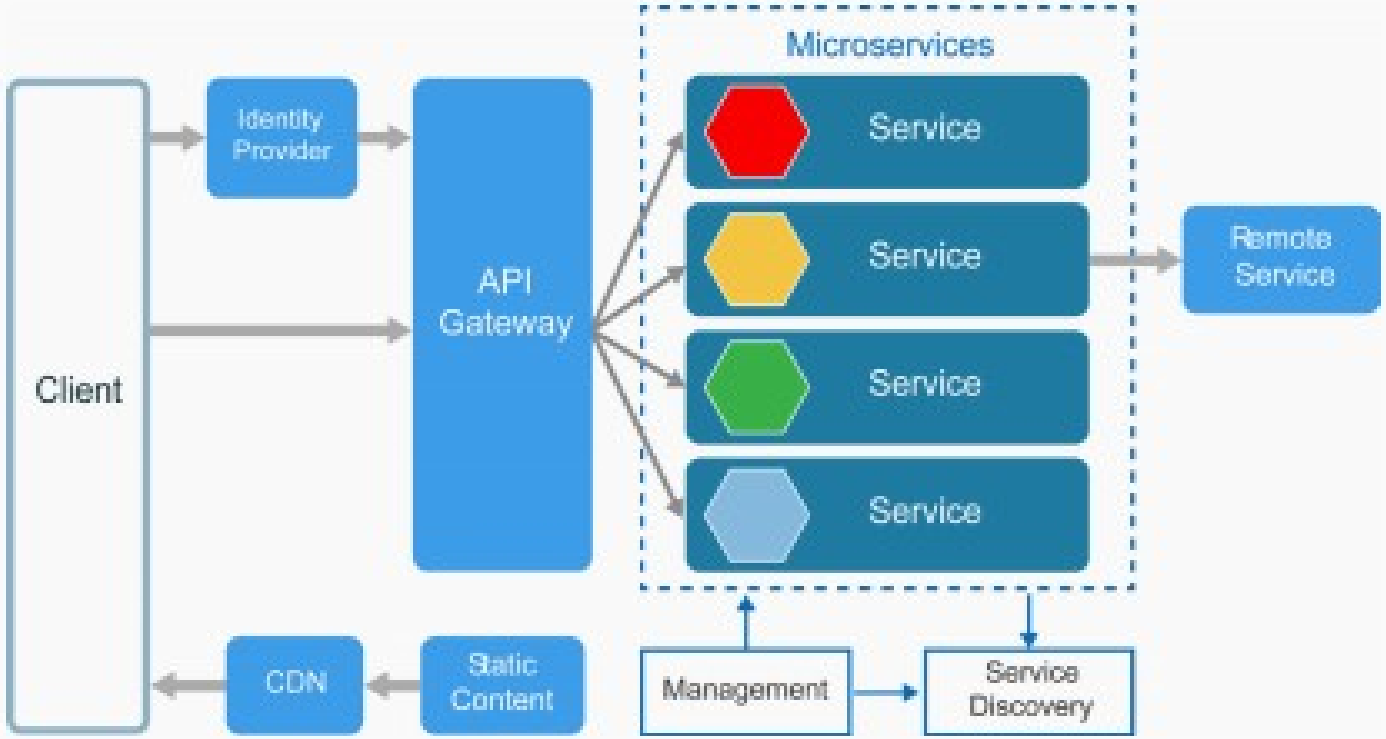


Retries con Backoff Exponencial

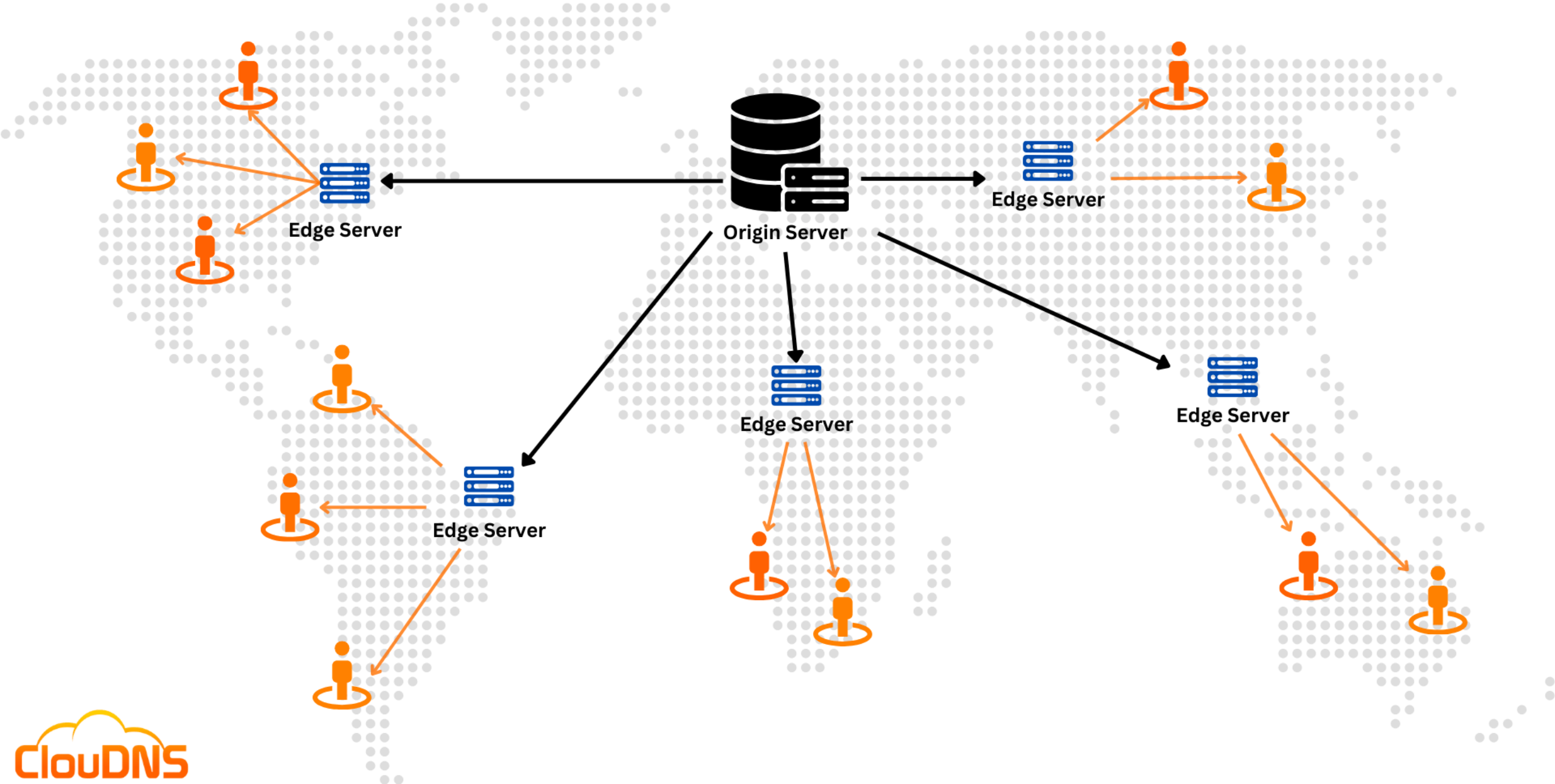
```
function retryWithBackoff(fn, retries = 5, delay = 1000) {
    return new Promise((resolve, reject) => {
        fn()
            .then(resolve)
            .catch(err => {
                if (retries > 0) {
                    const newDelay = delay * 2;
                    setTimeout(() => {
                        retryWithBackoff(fn, retries - 1, newDelay).then(resolve).catch(reject);
                    }, newDelay);
                } else {
                    reject(err);
                }
            });
    });
}

// Uso:
retryWithBackoff(() => fetch('https://example.com'))
    .then(response => console.log('Éxito:', response))
    .catch(err => console.error('Fallo:', err));
```

Seguridad


Microservices
DevOps
Client
API Gateway
Service
Service
Service
Service
Management / Orchestration


Microservices
Client
Identity Provider
API Gateway
Service
Service
Service
Service
Remote Service
CDN
Static Content
Management
Service Discovery

CDN (Content Delivery Network)



Conectores

“Un medio de comunicación entre componentes, que abstrae las interacciones desde llamadas locales hasta protocolos distribuidos”.

Bass, L., Clements, P., & Kazman, R. (2013)

Tipos:

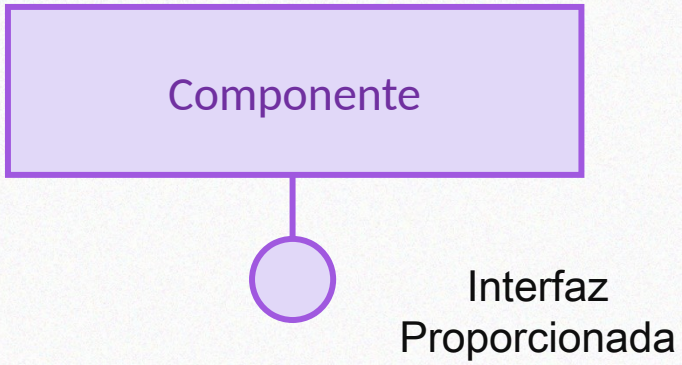
- **Sincrónicos:** RPC, HTTP.
- **Asincrónicos:** EventBus, colas de mensajes.
- **De transmisión:** Flujos de datos continuos.

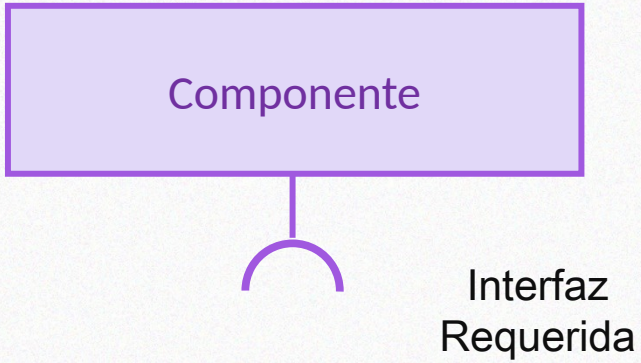
Interfaces

Las interfaces son elementos de componentes o clases que ofrecen funciones a otros componentes o clases. Definen operaciones que otros componentes o clases deben implementar.

Existen 2 tipos de interfaces:

- **Interfaces proporcionadas:** Servicios que el componente ofrece a sus clientes.
- **Interfaces requeridas:** Servicios que el componente necesita de otros componentes.



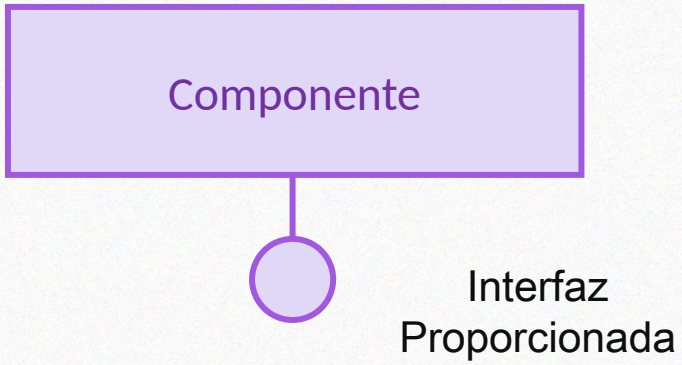


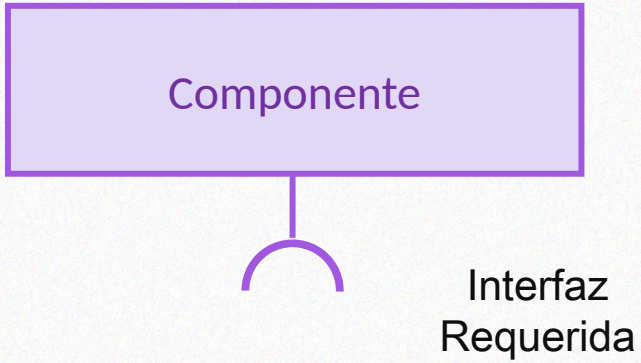
Interfaces

Las interfaces son elementos de componentes o clases que ofrecen funciones a otros componentes o clases. Definen operaciones que otros componentes o clases deben implementar.

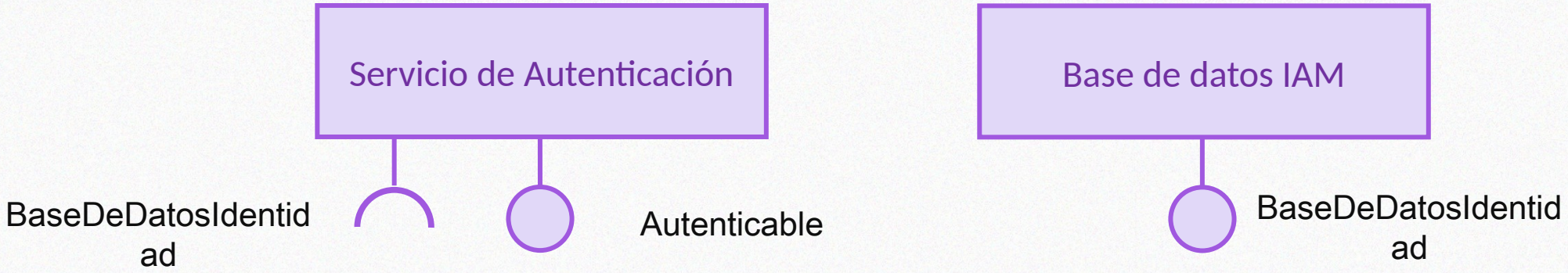
Existen 2 tipos de interfaces:

- **Interfaces proporcionadas:** Servicios que el componente ofrece a sus clientes.
- **Interfaces requeridas:** Servicios que el componente necesita de otros componentes.





Interfaces


Servicio de Autenticación
Base de datos IAM
BaseDeDatosIdentid
ad
Autenticable
BaseDeDatosIdentid
ad

Interfaces

```
public interface BaseDeDatosIdentidad {
    String obtenerIdentidad(String usuario);
}
```

```
public interface Autenticable {
    boolean autenticar(String usuario, String
contraseña);
}
```

Interfaces

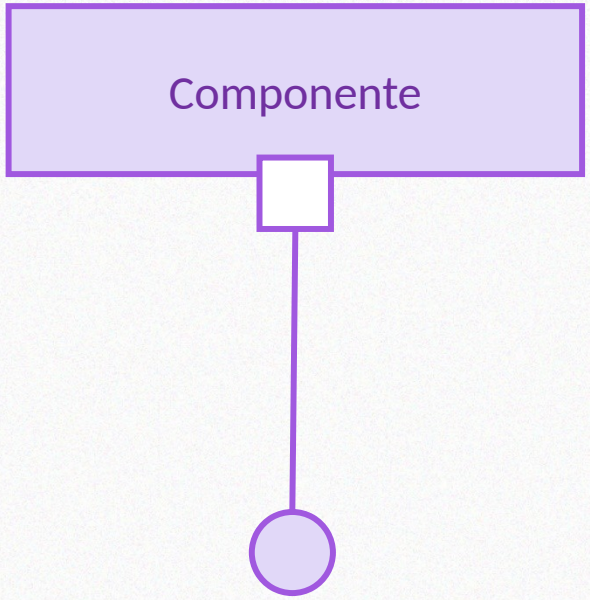
```java
public class BaseDeDatosIAM implements BaseDeDatosIdentidad {
    @Override
    public String obtenerIdentidad(String usuario) {
        return "juan".equals(usuario) ? "1234" : null;
    }
}
```

```java
public class ServicioDeAutenticacion implements Autenticable {
    private final BaseDeDatosIdentidad baseDeDatos;

    public ServicioDeAutenticacion(BaseDeDatosIdentidad baseDeDatos) {
        this.baseDeDatos = baseDeDatos;
    }

    @Override
    public boolean autenticar(String usuario, String contraseña) {
        String identidad = baseDeDatos.obtenerIdentidad(usuario);
        return identidad != null && identidad.equals(contraseña);
    }
}
```

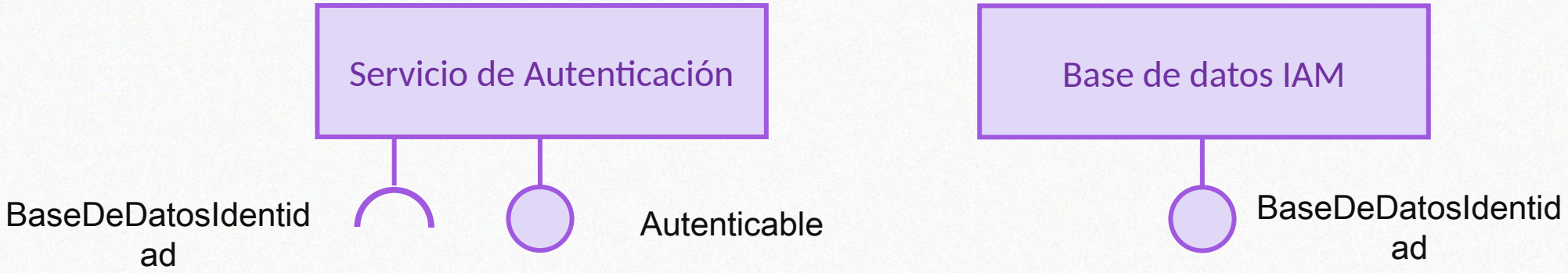
Puertos

Los puertos son lugares donde dos cosas pueden conectarse. Los símbolos para los puertos son pequeños círculos o rectángulos en el límite de un componente.

Permiten que los componentes se comuniquen entre sí. Por ejemplo, el componente de autenticación podría tener un puerto para recibir información de inicio de sesión del usuario.



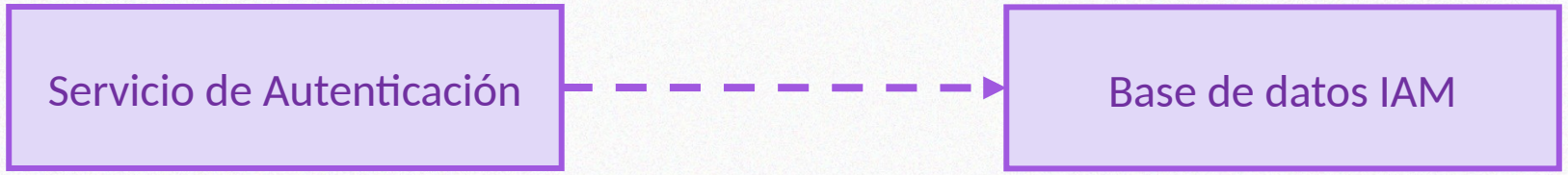
Interfaces

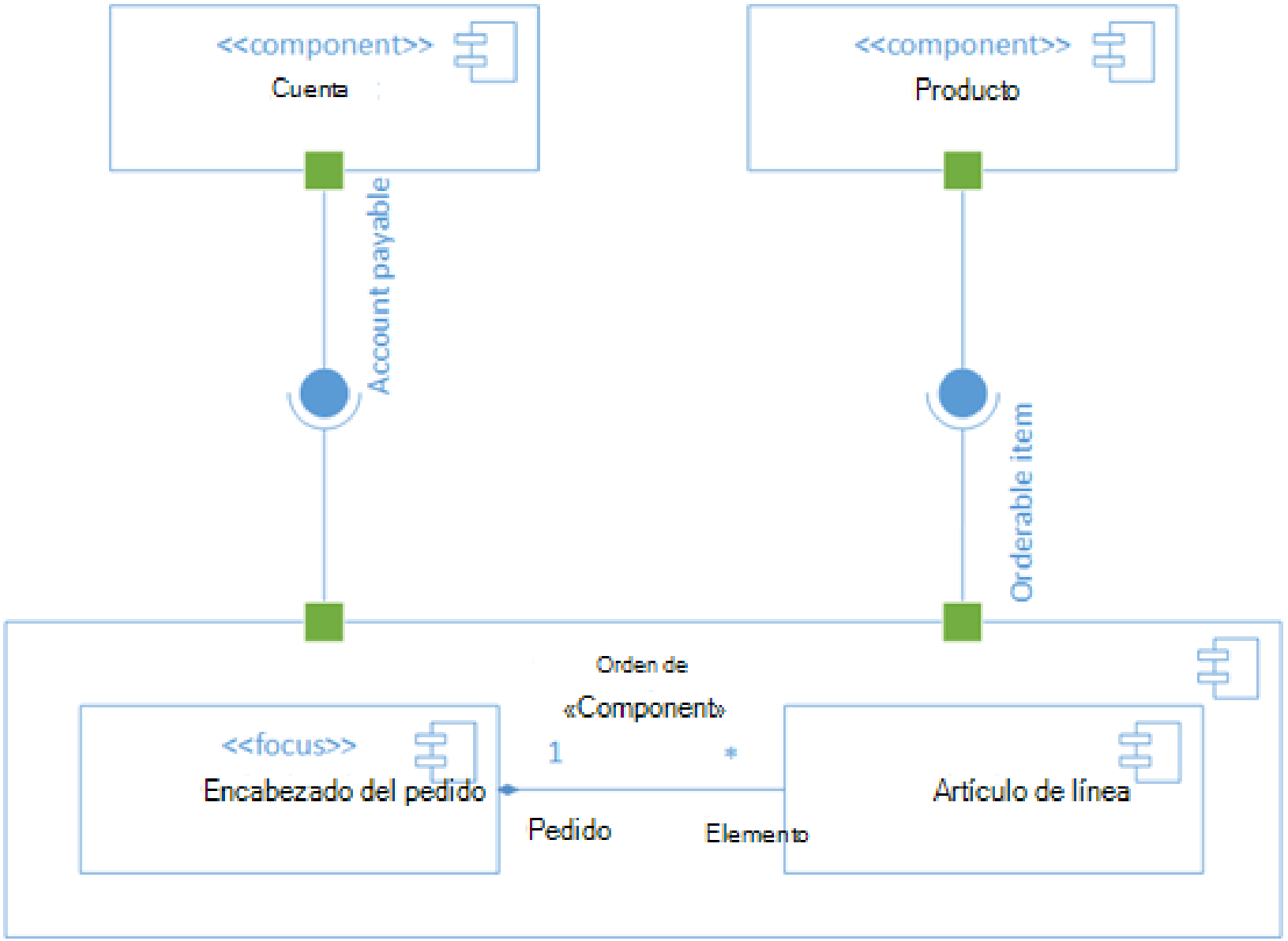

Servicio de Autenticación
Base de datos IAM
BaseDeDatosIdentid
ad
Autenticable
BaseDeDatosIdentid
ad

Puerto

```java
public class ServicioDeAutenticacion {

    private final PuertoAutenticacion puerto;

    public ServicioDeAutenticacion(BaseDeDatosIdentidad baseDeDatos) {
        this.puerto = new PuertoAutenticacion(baseDeDatos);
    }

    public Autenticable obtenerPuerto() {
        return puerto;
    }
    private class PuertoAutenticacion implements Autenticable {
        private final BaseDeDatosIdentidad baseDeDatos;

        public PuertoAutenticacion(BaseDeDatosIdentidad baseDeDatos) {
            this.baseDeDatos = baseDeDatos;
        }

        @Override
        public boolean autenticar(String usuario, String contraseña) {
            String identidad = baseDeDatos.obtenerIdentidad(usuario);
            return identidad != null && identidad.equals(contraseña);
        }
    }
}
```

Dependencias



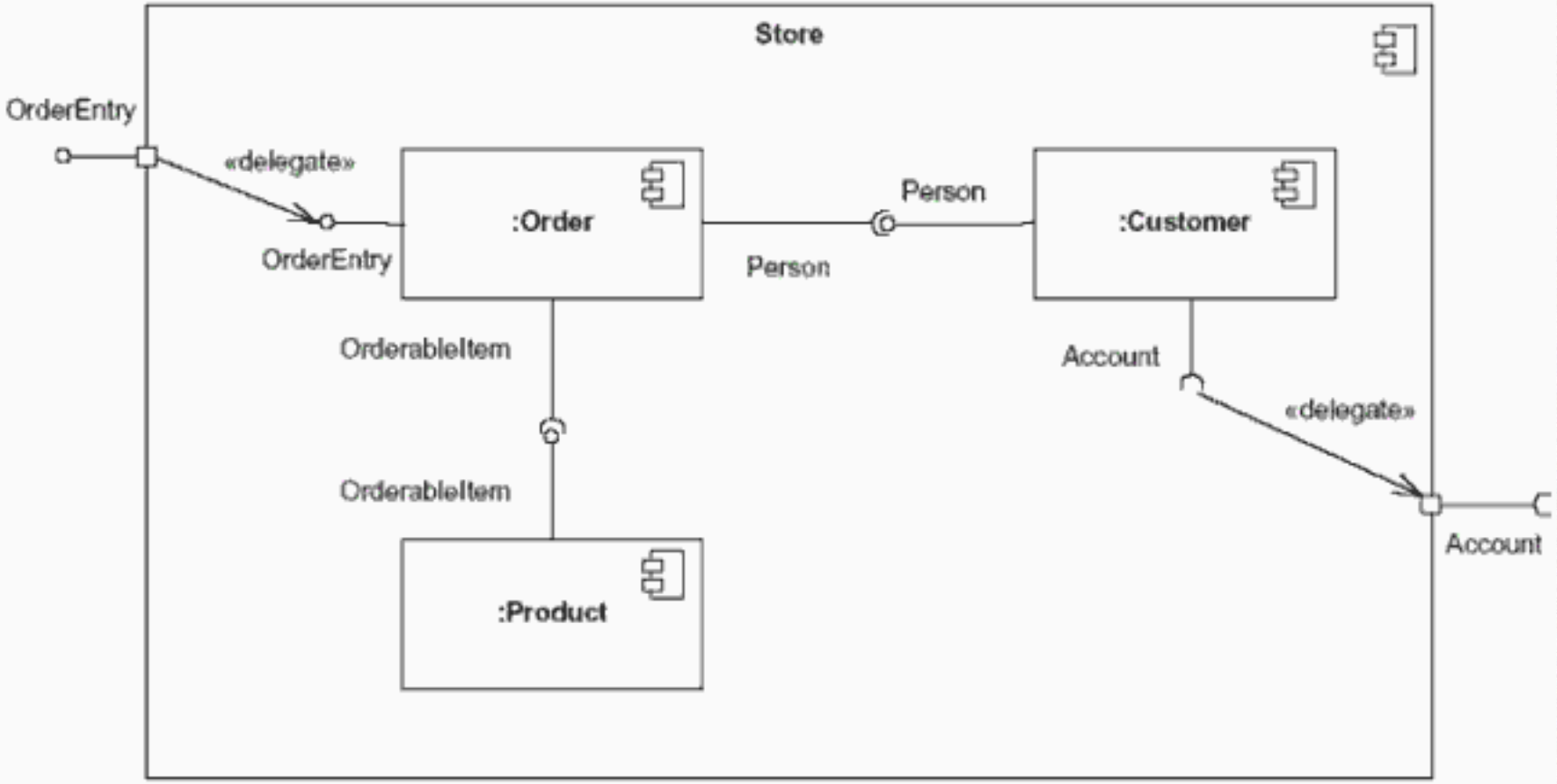

<<component>>
Cuenta
<<component>>
Producto
Account payable
Orderable item
Orden de
«Component»
<<focus>>
Encabezado del pedido
1
*
Pedido
Elemento
Artículo de línea

Conectores

Tipos de Conectores:

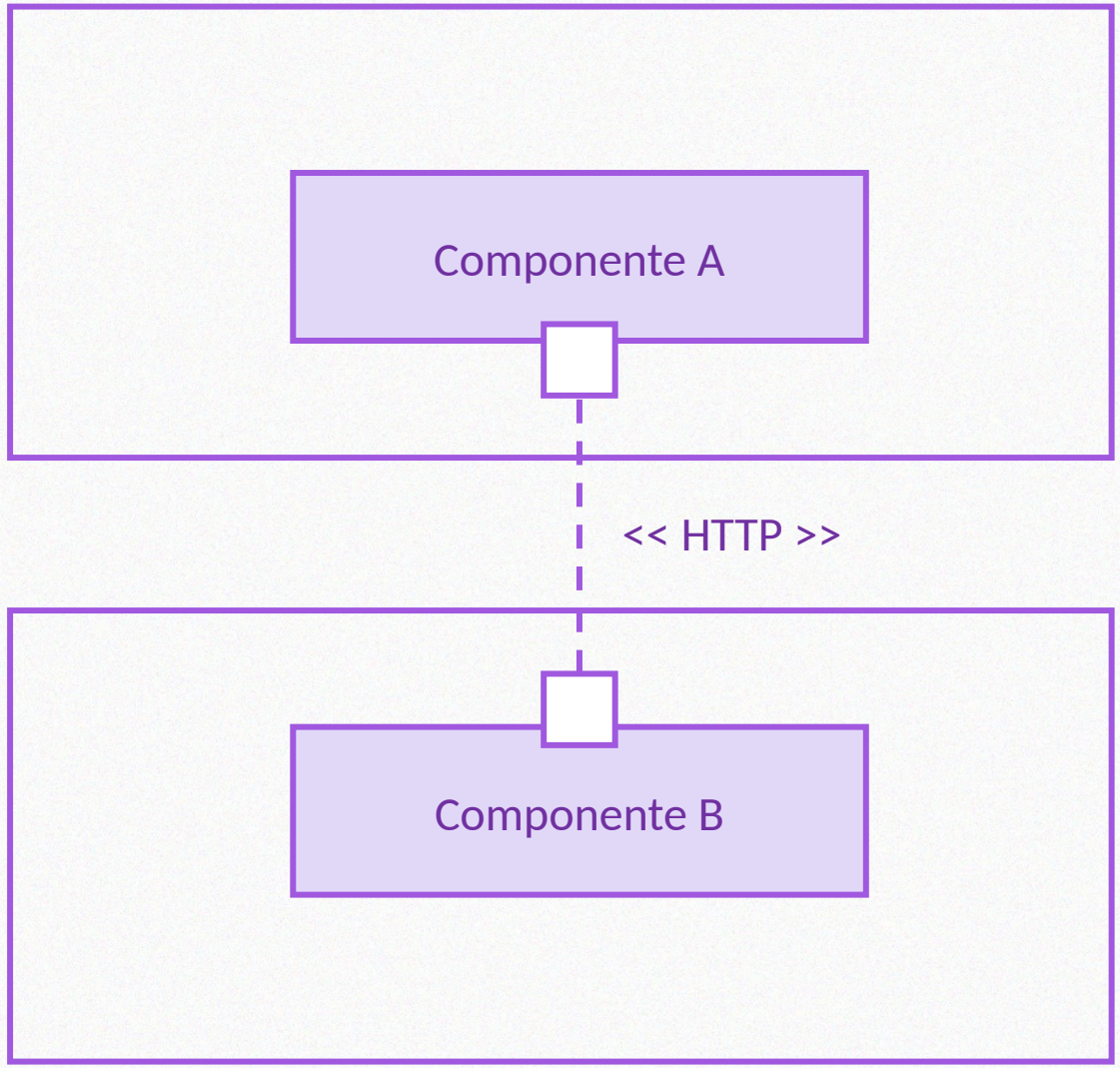
- **Sincrónicos:** RPC, llamadas directas (gRPC, HTTP en modo sincrónico).
- **Asincrónicos:** Mensajería basada en eventos (RabbitMQ, Kafka).
- **De transmisión:** Streaming (WebSockets, gRPC bidireccional).
- **Intermediarios:** Proxies, Gateways, Middleware.

Conectores


Store
OrderEntry
«delegate»
:Order
OrderEntry
Person
Person
:Customer
OrderableItem
Account
«delegate»
OrderableItem
Account
:Product

Conectores

Client-server - permiten que un conjunto de clientes concurrentes recuperen datos sincrónicamente a través de requerimientos de servicios



Conectores

```java
import spark.Spark;


public class ComponenteB {
    public static void main(String[] args) {
        // Configurar el puerto del servidor
        Spark.port(8080);

        // Endpoint HTTP que expone un servicio
        Spark.get("/data", (req, res) -> {
            res.type("application/json");
            return "{ \"message\": \"Hola desde Componente B\" }";
        });

        System.out.println("Componente B listo en http://localhost:8080/data");
    }
}
```

Conectores

```java
import spark.Spark;


public class ComponenteB {
    public static void main(String[] args) {
        // Configurar el puerto del servidor
        Spark.port(8080);

        // Endpoint HTTP que expone un servicio
        Spark.get("/data", (req, res) -> {
            res.type("application/json");
            return "{ \"message\": \"Hola desde Componente B\" }";
        });

        System.out.println("Componente B listo en http://localhost:8080/data");
    }
}
```

Conectores

```java
import spark.Spark;
public class PuertoServidor {
    public void start() {
        Spark.port(8080);

        Spark.get("/data", (req, res) -> {
            res.type("application/json");
            return "{ \"message\": \"Hola desde Componente B\" }";
        });

        System.out.println("PuertoServidor listo en http://localhost:8080/data");
    }
}

public class ComponenteB {
    public static void main(String[] args) {
        PuertoServidor puertoServidor = new PuertoServidor();
        puertoServidor.start(); // Iniciar el puerto proporcionado
    }
}
```

Conectores

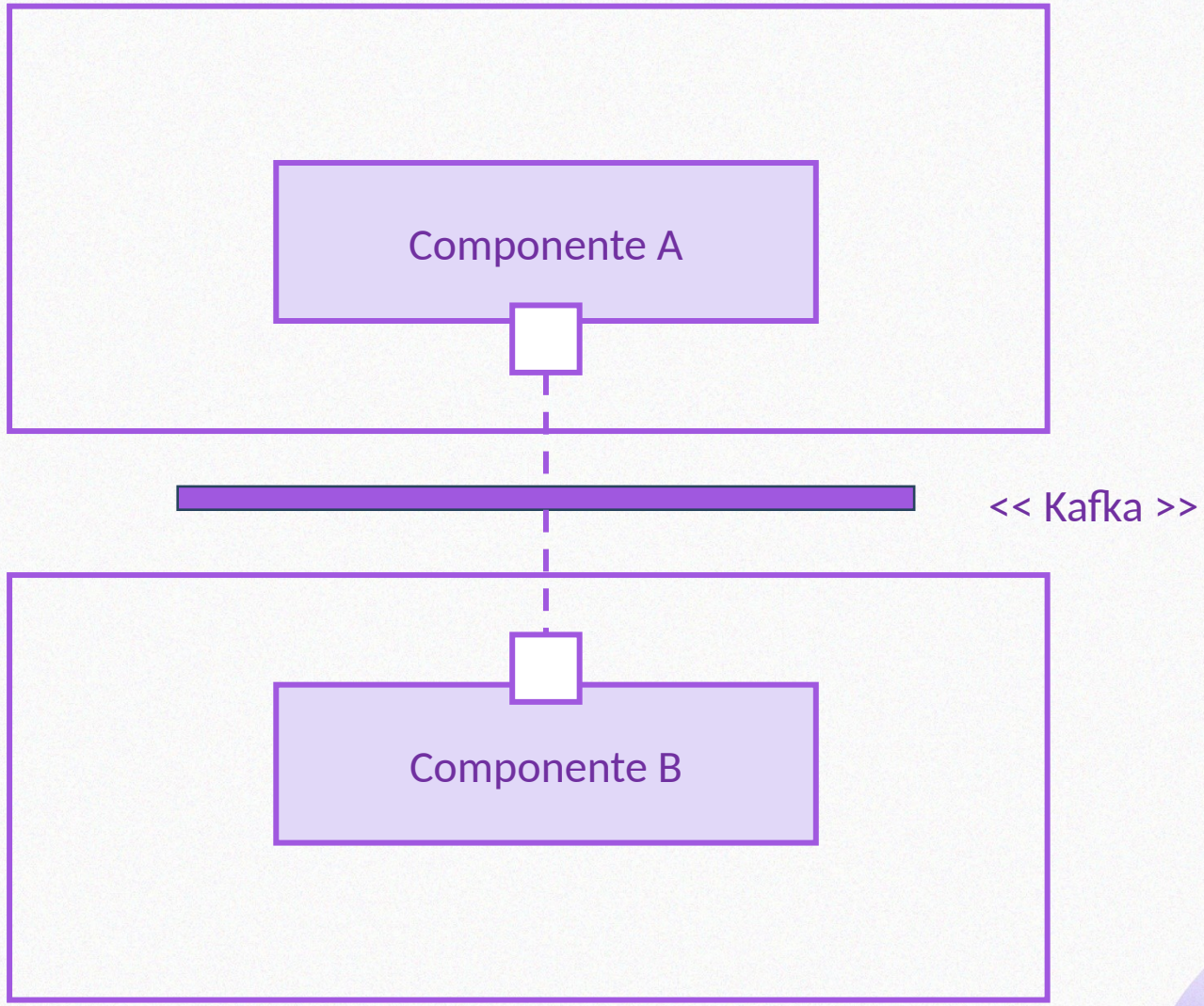
```
public class PuertoCliente {
    private final String baseUrl;

    public PuertoCliente(String baseUrl) {
        this.baseUrl = baseUrl;
    }

    public String obtenerDatos() {
        // Realizar solicitud HTTP GET al servidor
        HttpResponse<JsonNode> respuesta = Unirest.get(baseUrl + "/data").asJson();

        // Retornar el mensaje recibido
        if (respuesta.getStatus() == 200) {
            return respuesta.getBody().getObject().getString("message");
        } else {
            throw new RuntimeException("Error al conectar con el servidor: " +
respuesta.getStatus());
        }
    }
}
```

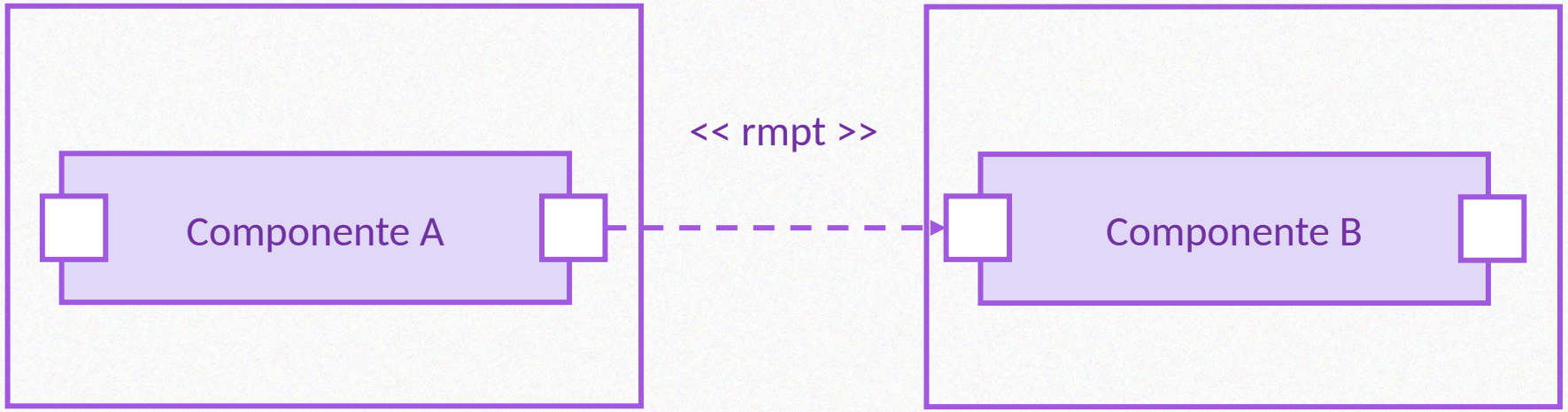
Conectores

```java
public class ComponenteA {
    public static void main(String[] args) {
        // Crear el puerto requerido
        PuertoCliente puertoCliente = new PuertoCliente("http://localhost:8080");

        // Usar el puerto para interactuar con el servidor
        String respuesta = puertoCliente.obtenerDatos();
        System.out.println("Respuesta del Componente B: " + respuesta);
    }
}
```

Conectores

Asincrono - Soporta envío de eventos y notificaciones asincrónicas.



Conectores

Transmision - Soporta el flujo continuo de informacion